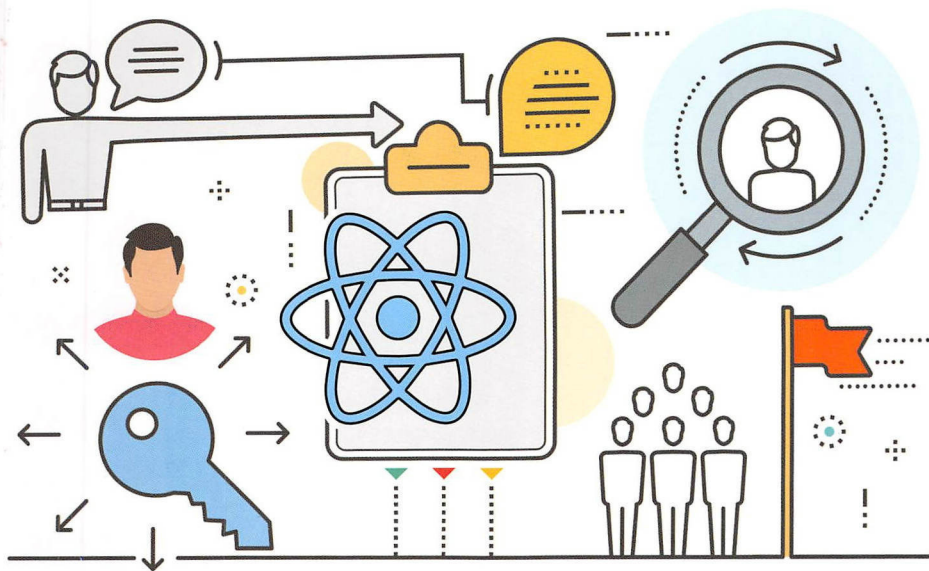


版权相关注意事项：

- 1、书籍版权归著者和出版社所有
- 2、本PDF来自于各个广泛的信息平台，经过整理而成
- 3、本PDF仅限用于非商业用途或者个人交流研究学习使用
- 4、本PDF获得者不得在互联网上以任何目的进行传播
- 5、如果觉得书籍内容很赞，请一定购买正版实体书，多多支持编写高质量的图书的作者和相应的出版社！当然，如果图书内容不堪入目，质量低下，你也可以选择狠狠滴撕裂本PDF
- 6、技术类书籍是拿来获取知识的，不是拿来收藏的，你得到了书籍不意味着你得到了知识，所以请不要得到书籍后就觉得沾沾自喜，要经常翻阅！！经常翻阅
- 7、请于下载PDF后24小时内研究使用并删掉本PDF



React

状态管理①同构实战

侯策 颜海镜 著



中国工信出版集团



电子工业出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
<http://www.phei.com.cn>



作者简介



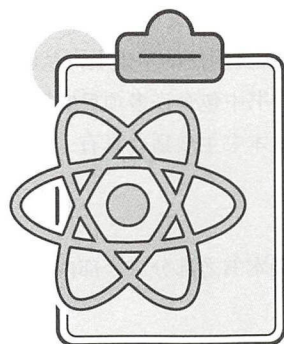
侯策

硕士毕业于法国国立高等电信学校。曾任职于BePATIENT集团，负责互联网+医疗平台的研发。曾任职于法国能源和苏伊士集团，参与欧洲天然气运输和费用系统的研发。2015年回国加入百度知识搜索部，负责多个产品线的大型技术迭代。行业之外是一名国家二级运动员（足球项目），曾组织过赴北非撒哈拉地区看望孤儿等慈善活动。



颜海镜

知名技术博主，开源达人，常以歪脖无脸男形象作为头像，经过多年沉淀，专注Web前端开发，先后任职于金山、百度、美团点评，负责前端开发工作。



React

状态管理^与同构实战

侯策 颜海镜 著

电子工业出版社

Publishing House of Electronics Industry

北京•BEIJING



内 容 简 介

React 自开源以来,便以革命性的设计理念迅速颠覆了前端开发的传统意义,其倡导的组件化、状态管理、虚拟 DOM 等思想极大提高了前端开发效率。为了更加高效地维护 React 应用的数据状态,以 Redux 为代表的数据库管理模式横空出世。

本书以 React 技术栈为核心,在介绍 React 用法的基础上,从源码层面分析了 Redux 思想,同时着重介绍了服务端渲染和同构应用的架构模式。书中包含许多项目实例,不仅为用户打开了 React 技术栈的大门,更能提升读者对前沿领域的整体认知。本书主要适合具有一定 JavaScript 基础的前端工程师,以及对前端开发感兴趣的相关从业人员阅读。

未经许可,不得以任何方式复制或抄袭本书之部分或全部内容。

版权所有,侵权必究。

图书在版编目(CIP)数据

React 状态管理与同构实战/侯策,颜海镜著. —北京:电子工业出版社,2018.8

ISBN 978-7-121-34554-8

I. ①R… II. ①侯… ②颜… III. ①移动终端—应用程序—程序设计 IV. ①TN929.53

中国版本图书馆 CIP 数据核字(2018)第 135288 号

策划编辑:孙奇俏

责任编辑:葛娜

印刷:三河市君旺印务有限公司

装订:三河市君旺印务有限公司

出版发行:电子工业出版社

北京市海淀区万寿路 173 信箱 邮编 100036

开本:787×980 1/16 印张:20.75 字数:449 千字

版次:2018 年 8 月第 1 版

印次:2018 年 8 月第 1 次印刷

定价:79.00 元

凡所购买电子工业出版社图书有缺损问题,请向购买书店调换。若书店售缺,请与本社发行部联系,联系及邮购电话:(010) 88254888, 88258888。

质量投诉请发邮件至 zlts@phei.com.cn, 盗版侵权举报请发邮件至 dbqq@phei.com.cn。

本书咨询联系方式:010-51260888-819, faq@phei.com.cn



推荐序 1

在前端开发这样一个重视界面开发的领域中，组件化几乎是“真理”。但是不同场景下的组件化，以及不同人眼中的组件化，可能很不一样。

我见过许多不同的组件形态，有 ExtJS 这种高度封装的富组件、jQuery UI 这种与页面 DOM 强相关的组件、Bootstrap 这种重样式轻行为的组件，以及各种不知名的公司团队自研的组件等。

但是从来没有一个组件框架能够像 React 一样受到那么多的关注，被那么多的开发者谈论，它在很大程度上改变了前端开发方式。

毫无疑问，React 是成功的。为什么这么说呢？也许是因为它的组件模型足够简单、易于理解；也许是因为它允许开发者把视图写在 JSX 中；也许是因为它为很多前端工程师带来了如 JSX、V-DOM、Flux、immutable、HOC、Fiber 等新的理念（虽然有些理念不是它创造的）；也许是因为它对同构的支持；也许是因为 React Native；也许，因为以上全部。

但无论如何，这些理由都不重要。我看到的是，在我的周围，无论是否使用 React，大家都在谈论和学习它，在进行框架选型的时候，它都是无法回避的选项之一。

我们平时在开发项目时，框架选型仅仅是一个开始，真正考验工程师能力的是如何在选定框架的理念及用法下，驾驭越来越高的业务复杂度。React 的上手难度并不小。我们需要对其理念和设计思想有足够深入的了解，结合自身的业务判断哪些是真正有用的，从而不陷入过度设计中。我们需要对其实现原理有足够深入的了解，避免陷入性能的瓶颈。

在这本《React 状态管理与同构实战》中，作者对设计思想、框架实现、应用实践都进行了较多的剖析，相信读者一定能够通过学习，快速将本书中的知识应用到项目开发中。

而我更希望看到的是，读者阅读过后能够在应用层面进行更深入的思考。比如，什么东西



IV React 状态管理与同构实战

是应该作为应用状态被管理的，在什么样的场景下应该使用怎样的组件间通信模式，在相同的应用场景下如果不用 Redux 而使用 MobX，应该怎么实现……将这些思考融会贯通，而不再依赖一本书，这才是真的收获。

董睿

百度资深前端工程师、EFE 核心成员、San 框架主要开发者

2018 年 6 月于北京



推荐序 2

1995 年, Brendan Eich 花了 10 天时间开发出一门脚本语言, 取名为 Mocha, 并将其集成到了 Netscape 浏览器中, 不久后这门语言被改名为 LiveScript, 意思是可以让网页充满动力。同年年底, 网景公司和 SUN 公司达成协议并获得了 Java 商标的使用权, 于是正式将这门语言更名为 JavaScript。

历史选择了 JavaScript, 使其成为目前浏览器唯一内置支持的语言。时至今日, JavaScript 已经不仅仅局限于为网页开发实现特效, 而是真正发展成了一门全功能的编程语言。

我从 2005 年开始接触网页开发, 经历了 Web 开发的“上古时代”。在 Web 1.0 时期, 我们开发出来的网页是给人“看”的, 此时流行 jQuery 这种用来处理浏览器兼容性的库, 以及像 Dojo、YUI、ExtJS 这种用来做 UI 的库。

随着计算机和浏览器性能的提升, JavaScript 的功能开始不再局限于实现简单的网页开发, 特别是 Ajax 的使用更是显著提升了用户体验, 这个时期被称为 Web 2.0。站在开发者和使用者的角度, 在 Web 2.0 时期开发出来的网页是给人“用”的。此时的 JavaScript 程序无论是从代码数量还是代码复杂程度上, 都是前所未有的。于是开发者们开始借鉴后端流行的 MVC 框架的思想, 随后又根据前端自身的特点改进了传统 MVC 模式, 并发展出了 MVP、MVVM 等新架构, 其中比较有代表性的有 Knockout.js、Backbone.js、Ember.js 等。

后来, React 发布了, 自那时起我成为一名坚定的 React 使用者。React 不仅仅是一个全新的框架, 更是一种新的思想。React 重新定义了前端 View 层的开发模式: $v = f(s)$, 其中 s 代表引用的状态 (state), v 代表 View, 而 f 则是一个把状态映射为 View 的纯函数。这个简单的公式代表了前端开发的一种模式: View 就是对状态的展示, 对于同一个 f 而言, 相同的状态永远对应相同的视图。

React 就是这里的 f , React 生态的不同库则代表着不同的 f , 比如 react-native、react-art、react-canvas、react-svg 等。

当我们写 `<TextBox color="red">` 时, 它既可以被 react-dom 渲染为一个 div 标签, 也可以在



VI React 状态管理与同构实战

服务器端被渲染为一个字符串，还可以被 `react-native` 渲染为原生的控件，甚至可以被渲染为 Word 中的一行文本、Excel 中的一个表格等。而这一切的魔法就源自 React 的思想。

但是 React 只是一个专注于 View 层的框架，它只负责把状态映射为视图，并不关注状态的来源和转换，因此在实际开发中，我们还需要关注“React 全家桶”中的 Redux。另外，同构应用可以让开发者只编写一套代码便可以既运行在服务端，又运行在客户端，充分结合两者的优势，并有效避免两者的不足。这也是 React 的一大优势。

虽然市面上关于 React 的书已经数不胜数，但是大多都是围绕着 React 框架本身的使用方法来讲解的，对于深入讲解 React 状态管理与同构应用的书却寥寥无几，而侯策和颜海镜的这本书正好可以弥补这一方面的不足。

几年前，由于机缘巧合，我认识了本书的作者之一颜海镜。颜海镜不仅是开源的狂热爱好者，也是国内最早学习并实践 React 的开发者之一。从我认识他起，他就一直在关注各种前端新技术，并开源了很多前端开发工具和库，这一点真的非常难能可贵！

如果你想实战 React 同构应用，或者想要深入全面地了解有关 React 状态管理的知识，相信这本《React 状态管理与同构实战》一定会给你很多启发。强烈建议各位读者细细品读。

迷渡 (justjavac)

Flarum 中文社区创始人

2018 年 6 月于天津



序 1

一本书的诞生，可以说既是偶然，也是冥冥之中的必然。

当电子工业出版社的孙奇俏编辑第一次联系我向我约稿的时候，恰逢 2017 年谷雨时节，雨生百谷，万物蓬勃，破土向生。于我个人而言，那段时间正是我回国加入百度，需要迅速积累技术经验的阶段，于是我便无知无畏地开始了近一年的写作旅程。

当我在寻找选题时，毫不犹豫地将目光聚焦到了前端开发方向。我相信每位开发者都能清醒地意识到：这个领域既收获着发展，也迎接着淘汰；它既有着与生俱来的混乱，也有着与这种混乱抗衡的秩序；它既批量产生需求与迭代，也制造了同等规模的迷茫与困惑。没错，短短几年时间，前端开发者就脱离了“刀耕火种”的原始时期。

伴随着 JavaScript 语言的不断演进，Node.js 强势崛起，HTML 5 等技术攻城略地，巨大的信息量和学习成本如潮水般涌来。然而，发展的“副作用”是让开发者感受到前所未有的陌生：一切都在加速向前，自己却只能目瞪口呆。但我想，每个人都不甘心做一个原地踏步的旁观者。

这本书的诞生，和试图挣脱这种无力感有关。因此我选取了这场前端“工业革命”中最具代表性的潮头宠儿——React。以 React 技术栈为主题，将自己学习过程中所见所感的点滴片段用一根主线串起来，不断拓宽思考的边界，吸纳社区智慧进行深度剖析。我想从最初的那些困惑出发，用解读源码、分析设计模式、结合实战案例的方式，探究框架或技术栈“全家桶”的设计思路以及存在意义；探究何为昙花一现的技术趋势，何为永恒持久的思想价值；探究怎样增加对技术的掌控，以避免在快速发展的风暴中随波逐流……

React 绝不仅仅是一个灵活、高效的视图层开发库。截至本书写作之时，v16.4.1 版本共有 20 个分支，其代码仓库中有近 1 万次 commits，94 次发布的背后是 1193 名贡献者的付出，还有 102694 个 stars 和 18568 个 forks，这些数字构建起了一个庞大的技术社区，其背后蕴含了海量的优秀设计思想。

在这本书的整个写作过程中，我也再一次感受到了以 React 为中心的状态管理及同构应用的魔力——我体会到了组件化和传统视图层开发的巨大区别，体会到了数据驱动和面条式操作



DOM 的不同，体会到了虚拟 DOM 和性能优化的奥秘，体会到了状态管理背后的精妙设计，体会到了 Redux 是发布订阅模式和函数式的结晶，体会到了同构应用和服务端渲染的背后是架构设计的螺旋式变迁，是对用户体验和性能的不断打磨和孜孜追求。总之，我体会到了为什么 React 技术栈能够脱颖而出，因此也真心希望这本书能够对各位读者有所启迪。

回想起来，本书大部分内容是在北京完成的：起笔于仲夏，经历过“帝都”雾霾弥漫的冬季，完成于如今农历五月初五的端午佳节。那么索性就以屈原《离骚》中的诗句来结尾吧，与读者共勉。

愿我们对技术永远秉承“亦余心之所善兮，虽九死其犹未悔”的追求，以及“路漫漫其修远兮，吾将上下而求索”的态度。

侯策

2018 年 6 月于北京



序 2

人生需要勇气

人类的每一次进步，技术的每一次发展，都源自对未知世界的探索，探索让我们发现了更大的世界。有人说探索需要好奇心，有人说探索需要想象力，而我觉得，探索最需要的是勇气，面对未知，只有勇敢的人才能迈出第一步。

在前端的世界里从来不缺乏有勇气的人，这些勇士们引领着前端技术的不断变革，技术更新了一代又一代，从以 jQuery 为代表的操作 DOM 时代，到以 Backbone 为代表的 MVC 框架时代，到以 AngularJS 为代表的 MVVM 框架时代，再到以 React 为代表的前端技术的新一代，前端领域发展空前繁荣，颇有百家争鸣、百花齐放的局面，而这一切都源自各位开发者的勇气。

如果你对上述提到的名词不了解，或者对当前火热的技术感觉迷茫，没关系，不要担心。其实人的天性就是依赖熟悉的环境，我也曾害怕恐惧，也曾对新技术畏首畏尾，但幸运的是，我突破了自己——曾经陌生的名词，如今都被我驾驭得很好，是勇气给了我力量！

关于我和 React 的故事，要感谢本书的另一位作者——侯策。他是最志同道合的同事和朋友，他也是一位非常优秀的勇士。他最先研究 React，并给我介绍了很多与 React 相关的知识，我们共同探讨，最开始我们仅仅是基于 React 做一些比较小的内部系统，现在 React 已经变成主要的技术栈了，极大提高了我的工作效率。关于我对 React 的理解，你可以通过阅读本书的内容来了解，因为我已经把我的想法总结好，融入书里了。

关于我和这本书的故事，还是要感谢侯策。你之所以能看到这本书，是因为他是一位勇气值爆棚的真正勇士。最开始侯策和我说要写一本书的时候，我是拒绝的，虽然我写了很多博客文章，也阅读了很多书籍，但我从来没有写过书，面对未知我有点犹豫，但是最后我还是决定要试一试，因为这次帮我战胜未知的，除了勇气还有友谊。感谢勇气和友谊，让我又一次挑战了自己，也欢迎大家关注我的博客（yanhaijing.com），那里有更多关于我的故事。

写书和写博客还是有很大差别的，整个写作过程是一个挑战和突破的过程，因此书中难免有纰漏，也欢迎大家批评斧正。如果说，面对未知时是勇气让我们迈出了第一步，那么接下来靠的就是坚持了。做任何事情，都是重在开始，贵在坚持。

颜海镜

2018年6月于北京

前言

本书内容

本书以 React 技术栈为核心，在介绍 React 用法的基础上，从源码层面分析了 Redux 思想，同时着重介绍了服务端渲染和同构应用的架构模式。全书共分 8 章，其中每一章的主要内容如下。

第 1 章 React 与前端

本章简单介绍了前端开发的历史发展，以及 React 的诞生故事，并对本书后面章节要介绍的知识进行了简单概述。

第 2 章 深入浅出 React

本章围绕组件介绍了很多 React 相关知识，包括组件的实现方式、组件的抽象、JSX 语法、组件的生命周期、组件的属性和状态、如何进行事件交互、组件间如何通信、如何组织组件、组件与 DOM 的关系等。

第 3 章 Redux 应用架构基础

本章介绍了 Redux 基础及用法，包括 reducer 函数的编写、派发 action 的设计，以及 store 状态的更新流程等。在此基础上，还介绍了一个极为重要的概念——函数式编程。本章在数据的不可变性操作上进行了较为深入的实践。同时因为应用开发需求复杂，对于异步处理场景，本章也介绍了 Redux 中间件的使用方法。

第 4 章 深入理解 Redux

本章深入剖析了 Redux 源码及本质，细致讲解了 Redux 原理，介绍了其实现思想、中间件的设计思想、react-redux 库的奥秘，以及在实际开发中的一些最佳实践，帮助读者对 Redux 有一个更高层面的认知。

第5章 揭秘 React 同构应用

本章介绍了基于 React 开发同构应用的技术实现。前后端的合作和分工、模式的变迁和不同模式的优缺点，将会是一个永恒的话题。React 为同构应用打开了一扇窗户。在 React 同构设计以及 Node.js 迅速发展的背景下，前端开发完全可以拥有更广阔的空间。

第6章 深入理解 React 技术内幕与生态社区

本章对 React 及 Redux 中的热点话题进行了探索，介绍了 React 设计理念和魔法、React 组件的组合和复用、React “轮子” 开发、简单的 React 库编写、Redux 数据结构优化和角色分析等内容。结合社区中的优秀思想，希望在读者受到启发的同时，也打开一扇 React 进阶的大门。

第7章 单页面应用代码分割

本章深入讨论了 React 技术中的代码分割问题。代码分割不仅仅关系到性能优化，它更是一种技术工程设计的体现，直接影响用户体验。本章围绕这个主题进行了梳理与总结，并通过一个单页面应用实例进行了演示。

第8章 React 应用性能优化

性能涉及方方面面，如前端工程化、浏览器解析和渲染、比较算法等。本章主要介绍了 React 框架在性能上的优劣、虚拟的 DOM 思想，以及在开发 React 应用时需要注意的性能优化环节和手段。同时，优化手段也在与时俱进，不断更新，需要开发者时刻保持学习。

联系作者

在本书的写作过程中，我们力图严谨，细心检查了书中所有的内容，尽最大努力排除错误。但由于水平有限，若您在阅读过程中发现错误，产生疑问，或者对本书有其他好的建议、意见，都可以联系我们，我们定会及时回复。

联系请发邮件：lucashou@yeah.net。

致谢

感谢电子工业出版社博文视点策划编辑孙奇俏的邀约，感谢责任编辑葛娜老师的辛苦付出，她们的专业态度是成书的有效保障。

感谢颜海镜老师的加入，其除了负责本书第 1 章和第 2 章内容的撰写，更是为本书提供了无私的技术支持，他也是我本人的良师益友。

感谢沈抖先生在百忙之中对本书寄予评价与推荐，从更高的视角和层次对前端行业进行了总结、展望。感谢董睿先生为本书所做的精彩推荐序，他的谦虚品行和对技术的深刻理解值得每一位工程师学习。感谢 justjavac、阮一峰、狼叔、小燭、顾轶灵等各位老师给予本书的评价与推荐。作为作者，我深感荣幸；作为晚辈，我将力争向行业专家们看齐。

感谢颜适、张昌敏等同学对本书进行的细心审校，他们对本书内容提出了很多好的建议。感谢百度搜索部、公关部对我写书这件事的支持。

最后感谢我的家人、朋友、同事，你们的支持和鼓励，是我做每一件事情的动力源泉。

读者服务

轻松注册成为博文视点社区用户（www.broadview.com.cn），扫码直达本书页面。

- 下载资源：本书如提供示例代码及资源文件，均可在 [下载资源](#) 处下载。
- 提交勘误：您对书中内容的修改意见可在 [提交勘误](#) 处提交，若被采纳，将获赠博文视点社区积分（在您购买电子书时，积分可用来抵扣相应金额）。
- 交流互动：在页面下方 [读者评论](#) 处留下您的疑问或观点，与我们和其他读者一同学习交流。

页面入口：<http://www.broadview.com.cn/34554>



目 录

第 1 章 React 与前端	1
1.1 前端简史	1
1.1.1 Web 1.0	2
1.1.2 Web 2.0	2
1.2 React 是什么	3
1.2.1 声明式	3
1.2.2 基于组件	4
1.2.3 一次学习, 多端受用	4
1.3 React 家族	4
1.3.1 React Router	5
1.3.2 Redux	6
1.3.3 React Native	6
1.4 本章小结	7
第 2 章 深入浅出 React	8
2.1 组件	8
2.1.1 createClass	8
2.1.2 Component	9
2.1.3 Functional Component	10
2.1.4 PureComponent	11
2.2 组件与系统	11
2.2.1 传统系统	12
2.2.2 React 系统	12
2.3 神奇的 JSX	12
2.3.1 createElement	13

2.3.2 JSX.....	14
2.4 组件的生命周期	14
2.4.1 挂载	15
2.4.2 更新	15
2.4.3 卸载	16
2.5 组件的属性和状态	17
2.5.1 属性	17
2.5.2 状态	19
2.6 组件和事件	22
2.7 组件通信	24
2.7.1 父子组件	24
2.7.2 爷孙组件	28
2.7.3 兄弟组件	29
2.7.4 任意组件	30
2.8 组件的抽象与复用	32
2.8.1 组件的设计与抽象.....	32
2.8.2 继承	33
2.8.3 组合	34
2.8.4 高阶组件	36
2.9 命令式与 DOM.....	38
2.9.1 ref.....	39
2.9.2 findDOMNode	40
2.9.3 dangerouslySetInnerHTML	41
2.10 本章小结	41
第 3 章 Redux 应用架构基础	42
3.1 Redux 究竟是什么	42
3.1.1 Redux 是库结合模式	43
3.1.2 Redux 和 React 的关系	43
3.2 Redux 设计哲学	44
3.3 函数式编程和纯函数	48
3.3.1 了解“函数是一等公民”	48

3.3.2	了解纯函数和副作用.....	49
3.4	Redux 基本使用和实践.....	51
3.4.1	初识 store.....	51
3.4.2	构造 action.....	52
3.4.3	使用 action creator（构造器）.....	53
3.4.4	使用 dispatch 派发 action.....	54
3.4.5	编写 reducer 函数更新数据.....	54
3.4.6	合理拆分 reducer 函数.....	55
3.5	Redux 开发基础实例.....	58
3.6	reducer 编写关键：不可变性.....	62
3.6.1	共享和不可变性.....	62
3.6.2	在 Redux 架构下保证不可变性.....	65
3.7	Redux 中间件和异步.....	73
3.7.1	应用 redux-logger 中间件.....	74
3.7.2	应用 redux-thunk 中间件.....	75
3.7.3	组合使用中间件.....	78
3.8	Redux 与 React.....	78
3.8.1	Redux 架构回顾.....	79
3.8.2	Redux 和 React 衔接点.....	80
3.8.3	使用 react-redux 库.....	82
3.9	实现计数器的四种方式.....	85
3.9.1	纯 React 实现.....	86
3.9.2	纯 Redux 实现.....	88
3.9.3	React+Redux 实现.....	90
3.9.4	React+Redux 并使用 react-redux 库实现.....	92
3.10	完成一个工程化实例.....	98
3.10.1	项目目录.....	99
3.10.2	组件划分.....	100
3.10.3	确定应用数据状态.....	101
3.10.4	数据状态和相关组件数据分配.....	103
3.10.5	action 和 reducer.....	104

3.10.6 使用 react-redux 库进行连接	107
3.10.7 使用中间件和接入 redux-devtools	109
3.11 本章小结	113
第 4 章 深入理解 Redux	114
4.1 Redux 源码探索——store 的实现	114
4.2 Redux 源码探索——combineReducers 的实现	118
4.3 dispatch 的改造——实现记录日志	121
4.4 dispatch 的改造——识别 Promise	124
4.5 糅合多种 dispatch	126
4.6 Redux 源码探索——中间件的秘密	131
4.6.1 源码剖析	131
4.6.2 写一个中间件的套路	135
4.7 再谈 Redux 设计思想	136
4.7.1 Redux 的特性和限制	137
4.7.2 Redux 生态	139
4.8 react-redux 究竟是什么	142
4.8.1 Provider 组件	142
4.8.2 connect 方法	143
4.9 本章小结	145
第 5 章 揭秘 React 同构应用	146
5.1 前后端架构设计和服务端渲染概念	146
5.1.1 前后端配合技术的演进	146
5.1.2 技术历史总是惊人的相似	149
5.2 同构应用	150
5.2.1 什么是同构	151
5.2.2 同构的优势和劣势	151
5.3 使用 React 和 Redux 实现同构应用	152
5.3.1 使用 renderToString 和 renderToStaticMarkup 方法	153
5.3.2 使用 renderToNodeStream 和 renderToStaticNodeStream 方法	154
5.3.3 Redux 搭配 React 实现服务端渲染	155

5.4	React 16 在服务端渲染上的惊喜.....	157
5.5	同构项目实战：基于 Node.js 的“渐进式”流渲染	158
5.5.1	项目背景和技术栈介绍.....	158
5.5.2	项目目录	158
5.5.3	代码实现	160
5.5.4	同构应用与浏览器端渲染优势对比.....	166
5.6	Next.js 设计理念和使用	168
5.6.1	Next.js 的极简理念	168
5.6.2	Next.js 设计思想	169
5.7	使用 Next.js 实现同构应用	172
5.8	本章小结	183
第 6 章	深入理解 React 技术内幕与生态社区	184
6.1	React 组件的组合和复用——高阶组件.....	184
6.1.1	高阶组件注意事项和编写原则.....	185
6.1.2	高阶组件从场景到应用.....	186
6.2	高阶组件和 render prop	193
6.2.1	Mixins 的问题	193
6.2.2	高阶组件和 Mixins	195
6.2.3	render prop 是什么	197
6.3	React 组件的组合和复用——Function as Child Component.....	198
6.4	React 组件的组合和复用——Children API	203
6.5	React “轮子”是怎样设计的	209
6.6	setState 异步带来的讨论和思考	216
6.6.1	setState 的同步和异步之争	217
6.6.2	让 setState 连续更新的方法	219
6.6.3	setState Promise 化的讨论和尝试	220
6.7	React 组件和 React element 到底是什么	221
6.8	实现一个简易的 React 库	227
6.8.1	准备工作	229
6.8.2	初见雏形	231
6.8.3	持续迭代	233

6.8.4 总结与思考	238
6.9 引入 Redux 的必要性及利弊	239
6.10 如何设计并应用 Redux connect.....	243
6.11 使用 selector 实现最佳实践	248
6.11.1 selector 使用实例	248
6.11.2 使用神奇的 reselect 类库.....	250
6.12 Redux store 数据结构扁平化及在 Twitter 中的实践	255
6.12.1 数据结构扁平化的优化方向和手段.....	255
6.12.2 Twitter 在基于 Redux 架构下的数据实践	263
6.13 React state 和 Redux state 的选取原则	266
6.14 本章小结	267
第 7 章 单页面应用代码分割.....	269
7.1 React 和代码分割	269
7.1.1 代码分割的意义和普遍做法.....	270
7.1.2 基于业务的代码分割.....	271
7.1.3 合理选择分割维度.....	273
7.1.4 合理选择加载时机.....	273
7.1.5 按需加载实现原理.....	276
7.2 Redux reducer 层面代码分割	278
7.3 代码分割工程实例	283
7.3.1 依赖和业务分离.....	284
7.3.2 按需加载组件	285
7.3.3 收益与总结	287
7.4 本章小结	288
第 8 章 React 应用性能优化	289
8.1 React 应用性能的秘密	289
8.1.1 性能到底指什么.....	289
8.1.2 正确理解 React 虚拟的 DOM 带来的优化	291
8.1.3 使用 React.addons.Perf 进行性能调试.....	294
8.2 提升 React 应用性能的建议	295

8.2.1	最大限度地减少重新渲染.....	295
8.2.2	Redux connect 方法隐藏的性能优化思想	297
8.2.3	inline function 的反模式	300
8.3	使用 PureComponent 保证开发性能.....	302
8.4	Redux 中间件和 Web Worker	308
8.5	本章小结	311

第 1 章

React 与前端

我曾在微博上说过“React 就是哪吒”，那么一个前端框架和哪吒有什么关系呢？其实我觉得二者有某些相似的地方。

哪吒家室显赫，拥有的神器多到需要三头六臂才能拿得过来，后来经历磨难，脱胎换骨，能够独当多面。

React 源自 Facebook，集多项特色于一身——组件化、声明式、虚拟 DOM、局部更新、状态机等，React 16 引入的 Fiber 架构更可谓脱胎换骨，得 Flux、Redux、immutable.js、React Router 助阵，如虎添翼。

随着 React Native 的出世和 React Canvas 的诞生，更是让 React 如同有了三头六臂一般，将传统前端突破到安卓端、iOS 端，大有一统全端的趋势。

目前三大框架 Angular、React、Vue.js 逐鹿中原，无论谁主沉浮，我都相信能“革了 React 命”的一定不是 React 的仿制品。

如果你对上面提到的这些名词不是特别了解，建议你阅读本章后续的内容，本章将介绍前端的历史和 React 相关知识。

1.1 前端简史

在没有 Web 时，信息的传递不太容易，得发明一种工具，让信息的创造者和阅读者都能看到一样的东西，于是 Web 诞生了。

1.1.1 Web 1.0

Web 的内容由网站制作者生成，用户只能浏览内容，信息的流动方向只能从服务端到客户端，这一阶段被称为 Web 1.0 时代。有人说这个时代的 Web 是静态的，这一阶段主要涉及的技术是 HTML、CSS、JavaScript。

如果一段文字没有标点，那就不太容易阅读，标点其实是一种数据结构，数据必须有结构承载才能更容易传递。在 Web 中，HTML 就是数据的结构，比如标题、段落、强调、表格等，现在我们称其为语义化。

因为视觉是 Web 的主要传播途径，除了结构，视觉也是一种信息，CSS 成为视觉信息的载体，有 HTML 和 CSS 已经可以制作出页面了。由于 HTML 中大量使用 div 标签，HTML+CSS 也被称作 DIV+CSS。

光有视觉页面还是不够的，页面还要能和用户交互，比如点击按钮要有弹窗，JavaScript 正是用来做这部分工作的，被称为行为。至此，Web 三要素，结构、表现和行为就全了，分别对应 HTML、CSS 和 JavaScript。

HTML、CSS 和 JavaScript 要分离，一直被视为前端的金科玉律，CSS 选择器成为连接 HTML 和 CSS 的纽带，而 DOM 成为 HTML 和 JavaScript 的桥梁，CSSOM 承担了 JavaScript 和 CSS 的媒介。

1.1.2 Web 2.0

Web 的内容主要由用户生成，用户浏览其他用户创造的内容，这一阶段统称为 Web 2.0 时代。这一阶段涉及的技术繁多。

服务器语言和表单的发明，第一次让信息可以从客户端流向服务端，账号体系把用户从屏幕前搬到了 Web 上，这一阶段的技术栈没啥变化，JavaScript 也只是作为玩具语言，用来验证表单，前端开发者都被称作美工。

2006 年谷歌推出了 Gmail，可以不用刷新完成各种操作，其媲美客户端的体验，也将 Web 2.0 推上了浪潮之巅。

一种被称为 AJAX 的名词被发明——页面需要更新，JavaScript 向服务器发起请求，服务端不再返回页面，而是返回 XML 格式的数据，然后 JavaScript 将数据渲染到页面中——最开始采

用的是拼接字符串的形式，后来发明了前端模版，比如 `template.js`。

这一阶段涌现了大批技术，如 `prototype.js`、`Dojo`、`ExtJS`、`jQuery`，其巅峰是 `YUI 3`。这些工具各有各的优点，其中 `jQuery` 可以作为代表，其主要功能是抹平浏览器差异，简化 `DOM` 操作。

`JavaScript` 代码规模越来越大，一个 `JavaScript` 文件已经不能适应现状了，需要拆分成多个 `JavaScript` 文件（分治思想），代码之间存在依赖关系，且依赖关系越来越复杂，为此社区进行了各种尝试来解决这个问题，比如采用 `AMD`、`CMD`、`CommonJS`。2015 年 `ECMAScript 6` 定稿，带来了原生的模块系统，这一问题才最终被解决。

随着项目工程越来越大，代码的组织结构是不是也需要复用？有人将后端的 `MVC` 模式带到前端，但 `MVC` 并不适合前端，因此前端做了适当改变，诞生了 `MVP`、`MVVM` 等框架，典型的有 `Backbone.js`、`Knockout`、`AngularJS` 等。

2013 年发布的 `React`，将一种全新的思路带到了我们面前，一个新的时代已经来了。

1.2 React 是什么

传说 Facebook 的大神们，对现有的框架都不满意，于是有了 `React`。`React` 是一个用于构建用户界面的 `JavaScript` 库，和其他 `MVC` 库不一样的地方是，`React` 仅仅涉及界面层，类似于 `MVC` 中的 `View`。

`React` 是一次完整的抽象，改变了我们思考、设计和写代码的方式；`React` 是一次完整的统一，统一了以前很多种编写界面的方式。原生前端实现界面太过灵活，基本上团队里每个人都有自己的一套方法，而 `React` 是一套非常优雅的方法论，我们苦苦追寻了多年的最佳思想，竟然都在 `React` 里了。

`React` 有三个特色，分别是声明式、基于组件和一次学习，多端受用。

1.2.1 声明式

`React` 改变了也统一了界面的实现方式，在 `React` 中，将界面抽象为状态和视图，只需定义好每个状态对应的视图就行了，剩下的 `React` 会帮你搞定，比如状态改变时会自动刷新视图等。

下面代码中的 `flag` 如果发生变化，界面会自动刷新。

```
class HelloMessage extends React.Component {
```

```
render() {  
  const { flag } = this.state;  
  return (  
    <div>  
      {flag === 1 ? 'hello' : '哈喽'}  
    </div>  
  );  
}
```

1.2.2 基于组件

React 改变了写前端的习惯, 在 React 的世界里一切都是组件, 以前的入口在 HTML, 现在的入口在 JavaScript; 以前是 HTML + CSS + JavaScript 的组合, 现在是 JavaScript + CSS 的组合; 以前要想复用功能得拷贝三处代码, 现在仅需引用一个组件即可。来看一个组件的例子。

```
class HelloMessage extends React.Component {  
  render() {  
    return <div>Hello {this.props.name}</div>;  
  }  
}
```

上面的代码中包含两种组件, 一种是 HTML 标签组件, 另一种是 React 组件, 其区别是 HTML 标签以小写字母开头, React 组件以大写字母开头。

1.2.3 一次学习, 多端受用

React 中抽象了一层虚拟 DOM, 所以我们可以频繁地修改状态, 但是更改的都是虚拟 DOM。当虚拟 DOM 发生变化后, 会集中更新到真实 DOM, 因为虚拟 DOM 的存在, 只要替换掉底层的渲染引擎, 就可以突破浏览器了。React Native 就是将 React 实现到了原生 App, React 的一切都在, 但是底层却不是 DOM 了, 而是原生的 View。类似的还有服务端渲染, 这也是本书主要讲的内容。理解了 React 的思想, 就可以搞定 Web 和 App, 简直不能再棒了。

1.3 React 家族

React 只是视图层, React 家族还有一些其他成员, 本节来简单介绍一下。

1.3.1 React Router

传统 Web 都是多页面的，每个页面一个 URL，页面之间通过超链接跳转，由浏览器负责管理页面的跳转、前进、后退等功能，通过指定 URL 可以直接跳转到指定页面。

比如有一个首页和一个文章页，此时首页是一个 URL，文章页是一个 URL，可以从首页跳转到文章页，然后再从文章页跳转到首页，这一切都由浏览器完成（见图 1-1）。

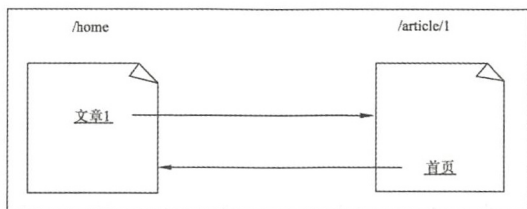


图1-1 路由跳转示意图

传统页面刷新跳转的方式，体验不是很友好，于是更友好的单页面应用来了，也被称为 WebApp。在这种模式下整个应用只有一个传统的页面，传统模式下的多个页面被抽象成一个个视图，原来的页面跳转，此时由 JavaScript 负责切换视图；原来向后端请求整个页面，现在变成向后端请求数据接口，因为不需要刷新页面，体验非常好。

但这种方式有一个问题，就是除了首页，其他页面是无法直接到达的，也就是每次都要先进到首页；为了能够实现传统 Web 那种 URL 的优点，需要在前端模拟一套路由，有了路由就可以通过 URL 直接进入某个视图了。

React 非常适合做单页应用，React Router 就是专门为 React 定制的路由，对 React 非常友好。下面看一下使用 React Router 实现上面首页和文章页跳转的情况。

```
import { render } from 'react-dom'
import { Router, Route, hashHistory } from 'react-router';

class App extends React.Component {
  render() {
    return (
      <div>
        {this.props.children}
      </div>
    )
  }
}
```



```
    }  
  }  
  render((  
    <Router history={hashHistory}>  
      <Route path="/" component={App}>  
        <Route path="home" component={Home} />  
        <Route path="article" component={Article} />  
      </Route>  
    </Router>  
  ), document.querySelector('#container'));
```

1.3.2 Redux

面向界面的编程可以分为三部分：界面、数据和数据操作。React 对界面的抽象做到了极致，但对数据和数据操作几乎没有约束，我们可以把这部分写到 React 组件中，也可以抽出来，将界面和数据与对数据操作进行分离，这就是 Redux 的工作。

Redux 是 JavaScript 的状态容器，提供可预测化的状态管理，Redux 因 React 而生，但也可以与其他类库配合使用。

1.3.3 React Native

React Native 是一套披着 React 外衣的原生控件，React Native 将原生控件封装为跨平台的 React 组件，并赋予我们通过 JavaScript 调用原生控件的能力。

在 React Native 里没有 CSS，但 React Native 让我们可以通过 CSS 的语法来设置原生控件的属性。下面是一个例子。

```
var React = require('react-native');  
  
var {Text, View} = React;  
  
var styles = React.StyleSheet.create({  
  container: {  
    color: 'red'  
  }  
});
```

```
class Hello extends React.Component {  
  render() {  
    return (  
      <View styles={styles.container}>  
        <Text>hello</Text>  
      </View>  
    )  
  }  
}
```

可以看到，React Native 就是用 React 语法封装过的原生控件，用 CSS 语法设置控件属性。

1.4 本章小结

本章简单介绍了前端的历史和 React 的诞生记，并对本书要讲的知识点做了简单介绍。本书后面会进行更深入的介绍，如果感兴趣就赶紧往下阅读吧。

第2章

深入浅出 React

本章将讲解 React 的多个知识点，主要围绕组件展开，包括组件的实现方式、组件的抽象、JSX 语法、组件的生命周期、组件的属性和状态、如何进行事件交互、组件间如何通信、如何组织组件、组件与 DOM 的关系等知识。

这既不是入门教程，也不是语法参考，我希望把最常用的 React 知识分享给你，包括我在学习 React 中的总结和思考，学会这些知识在平时的工作中将能够做到事半功倍。

2.1 组件

组件是 React 的核心，也是 React 的精髓。组件有输入、自己的状态和输出，输入在 React 中叫作 props，自己的状态在 React 中叫作 state，输出在 React 中是 render 函数返回的值（见图 2-1）。

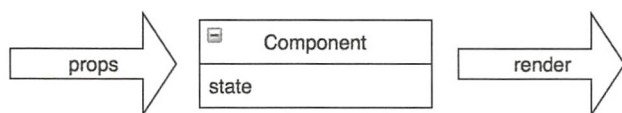


图2-1 组件构成示意图

组件必须先定义，然后才能使用。React 组件的定义方式也经历过几次进化，如果不了解，可能会造成困扰，下面逐一介绍一下。

2.1.1 createClass

这是最原始的方式，在比较早的书籍或文章中比较常见，现在已经不是很常用了。

```
var React = require("react");

var Hello = React.createClass({
  propTypes: {
    name: React.PropTypes.string
  },
  getDefaultProps: function () {
    return {name: 'yan'}
  },
  getInitialState: function () {
    return {count: 1}
  },
  render: function () {
    return <div>hello {this.props.name} {this.state.count}</div>;
  }
});
```

上面的代码会创建一个 Hello 组件，其中涉及了属性、状态和 render 函数。

创建组件后就可以将其渲染到 DOM 中。

```
var ReactDOM = require('react-dom');
ReactDOM.render(
  <Hello />,
  document.getElementById('root')
);
```

2.1.2 Component

随着 ES 6 的横空出世，React 也向 ES 6 靠拢，组件的定义不再使用 createClass 函数，而是改用 class 关键字。

上面的代码改用 ES 6 的方式书写会简单很多，也更容易理解。

```
import React from 'react';

class Hello extends React.Component {
  static propTypes = {
    name: React.PropTypes.string
```

```
    }  
    static defaultProps = {  
      name: 'yan'  
    }  
    constructor(props) {  
      super(props);  
      this.state = {count: 1};  
    }  
    render() {  
      return <div>hello {this.props.name} {this.state.count}</div>;  
    }  
  }  
}
```

2.1.3 Functional Component

有些组件有自身状态，比如交互类组件；有些组件没有自身状态，比如纯展示类组件。

然而，上面的方法并没有对两种组件做出区分，没有自身状态的组件，被称为无状态组件（stateless），对于无状态组件上面的方法有些影响性能，于是 Functional Component（函数式组件）横空出世。

```
import React from 'react';  
  
function Hello({name}) {  
  return <div>{name}</div>  
}  
  
Hello.propTypes = {  
  name: React.PropTypes.string  
};  
  
Hello.defaultProps = {  
  name: 'yan'  
};
```

函数式组件就是一个普通函数，其参数是一个对象，在被调用时就是传入的 props，配合函数参数的解构使用起来非常优雅。函数式组件的属性默认值和属性类型只能通过函数的属性定义。

其实 class 定义的类也是一个函数，函数式组件只是缺少了继承 React.Component。

2.1.4 PureComponent

PureComponent 是 React 15.3 引入的一个全新的组件基类，用来代替之前的 `react-addons-pure-render-mixin`。在 React 内部 PureComponent 继承自 Component，并将 `isPureReactComponent` 属性设置为 `true`。在 React 内部使用 `isPureReactComponent` 来区分是否是 PureComponent 组件。

PureComponent 和 Component 的功能几乎一样，但 PureComponent 的 `shouldComponentUpdate` 不会直接返回 `true`，而是会对属性和状态进行浅层比较，也就是仅比较直接属性是否相等。

可以用 `shouldComponentUpdate` 模拟 PureComponent，下面两个组件的功能一样。

```
class Demo1 extends Component {
  shouldComponentUpdate(nextProps, nextState) {
    const { props, state } = this;

    function shallowCompare(a, b) {
      return a === b || Object.keys(a).every(k => a[k] === b[k]);
    }

    return shallowCompare(nextProps, props) && shallowCompare(nextState, state);
  }
}

class Demo2 extends PureComponent {}
```

总结：如果环境支持 ES 6，那么应该使用 Component；如果环境不支持 ES 6，那么应该使用 `createClass`；如果组件没有自身状态，那么应该使用 Functional Component；如果组件是纯组件，那么应该使用 PureComponent。

2.2 组件与系统

React 带来了新的思想，也带来了新的挑战，页面的入口由 HTML 变成了 JavaScript，组成页面的最小单元由 HTML 标签变成了 React 组件。

React 带来了组件，但并没有告诉我们系统该如何构建，下面先总结传统的系统构建，再介绍使用 React 的系统构建。

2.2.1 传统系统

拿到一个新页面后，一般会先把页面划分成块，比如头部、尾部、侧边栏，然后分别完成每块内容的开发。在开发每个块时，先写 HTML，然后写 CSS，最后添加 JavaScript 逻辑。比较简单的页面就是一个 HTML、一个 CSS 和一个 JavaScript 文件，比较复杂的页面可能把 CSS 和 JavaScript 拆分成多个文件，上线时通过构建工具进行拼接。

随着时间的推移，页面会越来越多，会沉淀很多公共代码，比如公共的 CSS 片段、公共的 JavaScript 逻辑代码。公共代码一般分为两类，一类是纯逻辑功能，比如处理 URL 的代码；另一类是很多页面需要公用的业务代码，比如验证登录的代码。

但公共的 HTML 片段却不能被复用，比如每个页面的头部和尾部，因为 HTML 没有提供引入 HTML 片段的功能。一般对 HTML 片段的复用，都是通过后端语言进行的，比如 PHP 中的 include。

2.2.2 React 系统

传统系统最大的问题就在于作为页面入口和页面块入口的 HTML，竟然不能被引用。也就是说，在传统系统中要引用一个功能，需要分别引入 HTML、CSS 和 JavaScript 三部分，这对开发者来说非常不友好。而 React 就很好地解决了这个问题，因为页面的组成单元是组件，组件包括 HTML、CSS 和 JavaScript 三部分。

在 React 系统中，拿到一个新页面后，也是要把页面划分成块的，这里的块就是组件，然后把每个大组件拆分成更小的组件，接下来逐个组件进行开发，每个组件都包含逻辑和样式，最后把开发好的组件拼成页面就可以了。

在 React 系统中，除传统系统中公共的 CSS 片段和 JavaScript 片段之外，还有公共组件，其包括纯功能性组件和业务组件，页面头部就属于公共业务组件。

总结：React 组件让复用变得更容易，随着 Web Component 的发展，浏览器原生支持的组件，也许会让现在的组件写法发生一些变化。

2.3 神奇的 JSX

React 之所以能这么流行，JSX 功不可没，毫不夸张地说，没有 JSX，React 不可能这么流行。

在我看来，ExtJS 和 React 之间就差了一个 JSX，其实很多框架都差了一个 JSX。看似只差了一个简单的 JSX，其实差的是一棵语法树、一种编译能力、一种化繁为简的神器，其实能搞定语法树的程序员并不多。

在 React 中可以使用 createElement 和 JSX 两种方式来实例化组件，下面分别进行介绍。

2.3.1 createElement

在 React 中实例化组件使用 createElement，createElement 可以接收多个参数。

```
React.createElement(  
  type,  
  [props],  
  [...children]  
)
```

其中，第一个参数代表类型，可以是 HTML 标签或者 React 组件类型，HTML 标签都用小写字母表示，如 div、span 等；而 React 组件必须以大写字母开头，如 Hello。

第二个参数代表传给组件的属性，是一个对象。

后面的参数代表这个元素的子元素，每个子元素都可以是文本值或者 React 元素。

下面看一个 createElement 的例子。

```
React.createElement(  
  "div",  
  { className: "container" },  
  React.createElement(  
    Age,  
    { num: "1" },  
    "th"  
  ),  
  "Hello ",  
  this.props.name  
);
```

一眼看过去有些凌乱，最外面是一个 div 元素，其有一个属性 className，值为 container；其有三个子元素，分别是 Age、Hello 和 this.props.name。

我们用图来表示会更容易理解（见图 2-2）。

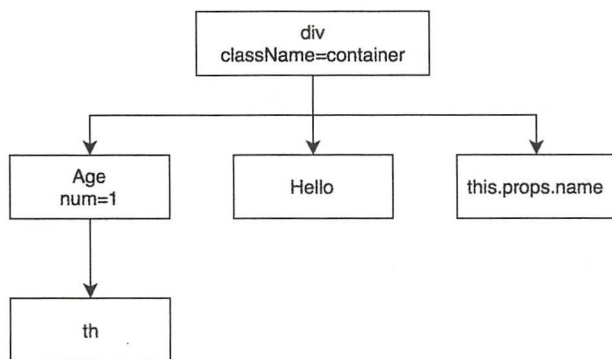


图2-2 代码样例中的React元素结构示意图

2.3.2 JSX

JSX 是一种扩展的 XML 语言，学习成本非常低。JSX 让我们以比较友好的方式，代替写各种函数调用、各种 new 操作，JSX 极大地简化了实例化组件的过程。

如果用 JSX 表示上面的逻辑，仅仅需要 4 行代码，同时也更容易理解，和前端熟悉的 HTML 非常相似。

```
<div className="container">
  <Age num="1">th</Age>
  Hello {this.props.name}
</div>
```

总结：JSX 最终要被编译为 createElement 才能够在浏览器里执行，平时写代码时一定要用 JSX，研究背后的原理一定要看 createElement。

2.4 组件的生命周期

每一个软件都会有诞生和死亡的一天，React 组件也有自己的生命周期，每一个 React 组件都会经历出生、存在和消亡的过程。在 React 的世界里，它们被称为挂载（Mounting）、更新（Updating）和卸载（Unmounting）。

React 为每个过程都提供了一些回调函数（称作钩子函数），让我们可以自定义一些内容。

2.4.1 挂载

在挂载过程中会依次执行下面的函数。

- constructor()
- componentWillMount()
- render()
- componentDidMount()

一般在 constructor 函数中初始化一些数据，比如设置 state 的初始值；componentDidMount 是最常用的回调函数，如果一个组件需要自己加载数据，一般都放到这个函数中。

```
class Hello extends React.Component {
  constructor(props) {
    super(props);
    this.state = {count: 1}; // 初始化 state
  }
  componentDidMount() {
    // 加载数据，成功后设置 state
    $.ajax({
      url: 'xxx',
      success: (ret) => {
        this.setState({count: ret.count})
      }
    });
  }
}
```

2.4.2 更新

在 React 中更新一个组件有三种途径，分别是父组件更新、自身状态变化、自身强制更新。

当父组件发生变化时，子组件需要重新渲染，此时会触发下面的函数。

- componentWillReceiveProps()

- `shouldComponentUpdate()`
- `componentWillUpdate()`
- `render()`
- `componentDidUpdate()`

当自身状态发生变化，也就是调用 `setState` 时，会触发下面的函数。

- `shouldComponentUpdate()`
- `componentWillUpdate()`
- `render()`
- `componentDidUpdate()`

调用 `forceUpdate` 会发生强制更新，此时会触发下面的函数。

- `componentWillUpdate()`
- `render()`
- `componentDidUpdate()`

`shouldComponentUpdate` 可以用来提升性能，`componentWillReceiveProps` 一般用来将新的 `props` 同步到 `state`。

2.4.3 卸载

当组件被卸载时会执行 `componentWillUnmount` 函数，一般会在这个函数里做一些清理工作，比如清除定时器、解绑自定义事件等。

在下面的例子中，如果不清除定时器，则会导致组件被卸载后，一直保留在内存中无法回收。

```
class Hello extends React.Component {  
  constructor(props) {  
    super(props);  
    this.state = {count: 1}; // 初始化 state
```

```
}  
componentDidMount() {  
  this.id = setInterval(() => {  
    this.setState({count: this.state.count + 1})  
  }, 1000);  
}  
componentWillUnmount() {  
  clearInterval(this.id);  
}  
}
```

总结：了解组件的生命周期可以快速解决很多问题，也可以在遇到问题时快速发现问题所在。

2.5 组件的属性和状态

组件根据不同的输入会有不同的输出，组件的输入在 React 中被抽象为属性；组件需要记录自身的一些变化，这一功能在 React 中被抽象为状态。本节就来讲一讲 React 中的属性和状态。

2.5.1 属性

当我们使用一个组件时可以传入一些属性，看起来和 HTML 的属性类似，也可以把属性想象成函数的参数。

```
<User name="yan" age="18">
```

如果 User 是一个函数式组件，则可以这样像下面这样使用传入的属性。

```
function User(props) {  
  return <div>{props.name}</div>  
}
```

如果 User 是一个 class 组件，则可以通过 this 引用属性。

```
class User extends Component {  
  render() {  
    return <div>{this.props.name}</div>  
  }  
}
```

需要注意的是，属性是只读的，即只能读取，不能进行更改。

18 React 状态管理与同构实战

有一个比较特殊的属性——`children`，代表当前组件的子组件集合。需要注意的是，自定义的属性名字不能和这个名字重复。在下面的例子中。可以通过 `this.props.children` 获取到三个 `li` 的内容，从而实现传入子组件的功能。

```
class List extends Component {
  render() {
    return <ol>{this.props.children}</ol>
  }
}

<List>
  <li>1</li>
  <li>2</li>
  <li>3</li>
</List>
```

组件的属性和函数的参数一样，如果使用者未传递属性值，则应该有默认值。ES 6 为 JavaScript 带来了原生的默认函数参数的功能，React 也提供了默认属性值的功能。

下面的例子如果引用 `User` 未提供属性值，就会使用默认属性值；否则会使用所提供的属性值。

```
class User extends Component {
  static defaultProps = {
    name: 'username',
    age: '0'
  }
}

<User>
<User name="yanhaijing" age="18">
```

对于公共组件，应该为属性添加默认值，这样组件才会更健壮，使用起来也会更简捷。对于业务组件，则建议为属性添加默认值。

有些时候并不是所有的属性都有默认值的，或者说有些属性不能省略，下面要讲的 React 的属性验证就可以解决这个问题。

JavaScript 是弱类型语言，也就是说，变量没有类型（值有类型），所以平时我们写的函数都需要对入参进行校验。下面的函数实现了两个数的相加，如果参数不为数字，则会打印一条

警告信息。

```
function sum(x, y) {  
  if (typeof x !== 'number' || typeof y !== 'number') {  
    console.warn('参数必须为数字')  
  }  
  
  return Number(x) + Number(y);  
}
```

其实平时我们写业务代码，可以省略对参数类型的校验。但如果是基础库的提供者，则必须要做入参校验，这样写出来的代码才能具有更好的兼容性、健壮性。

在其他强类型语言中，参数校验都是在编译过程中就做好了，但在 JavaScript 中必须手动做。为了简化这个流程，React 帮我们实现了一套属性验证方法。

下面的例子中如果 `age`，传入的属性不是数字类型，那么在开发模式下 React 会在控制台打印一条警告信息；如果 `name` 未传入属性也会打印一条警告信息。

```
import PropTypes from 'prop-types';  
  
class User extends Component {  
  propTypes: {  
    name: PropTypes.string.isRequired,  
    age: PropTypes.number,  
  }  
}
```

需要注意的是，在 React 15.5 之前，`prop-types` 并不是独立的，而是作为 React 的一部分，需要像下面这样使用。

```
import { PropTypes } from 'react';
```

2.5.2 状态

除了传入的属性，组件也有自身的状态。简单来说，就是刷新页面不会保持的信息都应该是组件自身的状态，比如下拉菜单是否展开就是自身的状态。

组件的状态都存储在 `state` 属性里，可以直接通过 `this.state` 读取。

```
class Clock extends Component {
```



20 React 状态管理与同构实战

```
render() {  
  return <div>{this.state.time}</div>  
}  
}
```

那么 `state` 中的数据是从哪里来的呢？一般都是在组件初始化的过程中，给 `state` 初始化默认值。下面两种方式的效果是一样的。

// 方式一

```
class Clock extends Component {  
  constructor(props) {  
    this.state = {  
      time: 0  
    }  
  }  
}
```

// 方式二（截至 2018 年 1 月 8 日，还不是语言标准，属于提案写法，不过 `babel` 已支持）

```
class Clock extends Component {  
  state = {  
    time: 0  
  }  
}
```

如果想要更新 `state` 中的数据，则需要使用 `setState`，当 `setState` 执行完成后，界面会自动更新。这也是使用 `React` 的一个好处，彻底分离界面和数据，并自动保持数据到界面的更新。

通过 `setState` 可以让上面时钟的例子，实现每秒自增的功能。

```
class Clock extends Component {  
  state = {  
    time: 0  
  }  
  componentDidMount() {  
    this.id = setInterval(() => this.setState({time: this.state.time + 1}), 1000);  
  }  
  componentWillUnmount() {  
    clearInterval(this.id); // 注意要清除  
  }  
}
```




```
render() {  
  return <div>{this.state.time}</div>;  
}
```

新手容易犯的一个错误是当 `setState` 执行完成后，去读取 `state` 中的数据时，却发现数据并未更新。

```
class Clock extends Component {  
  state: { time: 1 }  
  componentDidMount() {  
    this.setState({time: this.state.time + 1});  
    console.log(this.state.time) // 期待是 2，实际是 1  
  }  
}
```

这是因为为了提升性能，React 会把多次 `setState` 操作合并成一次，所以 `setState` 执行的过程是异步的。也就是说，`setState` 执行后并没有立刻更新 `state` 中的数据。那么如果想拿到 `setState` 执行后的数据，该怎么做呢？最简单的做法就是把计算结果存储下来。

```
class Clock extends Component {  
  state: { time: 1 }  
  componentDidMount() {  
    const newTime = this.state.time + 1;  
    this.setState({time: newTime});  
    console.log(newTime) // 期待是 2，实际是 2  
  }  
}
```

其实 React 也为我们提供了另一个方法——`setState` 还有第二个参数，它是一个函数，这个函数会在 `state` 更新后被调用。

```
class Clock extends Component {  
  state: { time: 1 }  
  componentDidMount() {  
    this.setState({time: this.state.time + 1}, () => {  
      console.log(this.state.time) // 期待是 2，实际是 2  
    });  
  }  
}
```



不要在 render 中使用 `setState`，其实这个问题一般不容易出现，因为大家都不这么写代码。但是还是要知道这一点，因为 `setState` 会触发 render。如果在 render 中再调用 `setState`，那么就会出现死循环，虽然 React 做了优化，不会卡死，但程序的响应会非常慢，所以一定不要这么做。

新手容易犯的另一个错误是，什么都放到 state 里，用了 React 感觉就把基本的 JavaScript 知识全忘了。只要这个数据不会影响到 UI 的变化，也就是数据变化不会引起 UI 变化，都没必要放到 state 中，以避免不必要的浪费。

一个基本原则就是，能放到文法作用域里的，能放到 this 里的，都不要放到 state 中。

总结：正确使用属性和状态，能够让程序设计更合理、性能更高、扩展性更好。

2.6 组件和事件

到目前为止，组件还仅仅能够展示和更新，不能和用户交互。下面来看一看如何给组件添加事件。

其实在 React 中绑定事件的方式和最初 HTML 绑定事件类似，但还是有一点区别的。

- React 中的事件名字是驼峰式的，而 HTML 中的事件名字必须全部用小写字母。
- React 中的事件处理器是一个函数，而 HTML 中的事件处理器是一个字符串。

下面分别用两种方式来实现同一个例子。

HTML 方式：

```
<button onclick="console.log('我被点击了')">来点我啊</button>
```

React 方式：

```
class Demo extends Component {
  onClick() {
    console.log('我被点击了')
  }
  render() {
    return <button onClick={this.onClick.bind(this)}>
      来点我啊
    </button>
  }
}
```



```
        </button>
      }
    }
  }
```

熟悉前端的读者初次接触 React 绑定事件的方式时一定会嗤之以鼻，这不违背了 JavaScript 和 HTML 分离的前端金科玉律吗？

下面来解答这个问题。在 React 中编写的代码其实是 JavaScript 代码，而不是 HTML，JSX 也是要编译成 JavaScript 的。React 其实是把 HTML 写到 JavaScript 中的，所以和在 HTML 中写 JavaScript 的方式还是有本质区别的。

下面来说说 React 事件的优点。

- React 事件是声明式的，让我们彻底绕过了选择器，绑定事件的过程变得非常简单。
- React 事件是天生的事件代理，看起来事件散落在元素上，其实 React 仅仅在根元素绑定事件，所有事件都通过事件代理响应。

React 也会给事件处理函数传入一个事件对象参数，这个参数的很多功能和原生事件对象很相似，比如可以阻止默认事件。

```
onClick(e) {
  e.preventDefault();
}
```

但需要注意，这个事件对象并不是原生事件对象，而是 React 基于 W3C 规范封装过的，屏蔽了浏览器差异。React 也提供了访问原生事件对象的方式。

```
onClick(e) {
  e.nativeEvent // 原生事件对象
}
```

在绑定事件时经常遇到的一个问题是，假如有一个列表，如果需要把列表的索引传给事件处理函数，数该如何做？办法有两种，一种是使用 bind。

```
class Demo extends Component {
  onClick(index) {
    console.log('index')
  }
  render() {
    return (
```



24 React 状态管理与同构实战

```
        <ul>
          {list.map((item, key) => (
            <li onClick={this.onClick.bind(key)}>item.name</li>
          ))}
        </ul>
      );
    }
  }
}
```

另一种办法就是包裹一层函数。

```
class Demo extends Component {
  onClick(index) {
    console.log('index')
  }
  render() {
    return (
      <ul>
        {list.map((item, key) => (
          <li onClick={() => this.onClick(key)}>item.name</li>
        ))}
      </ul>
    );
  }
}
```

在 React 中绑定事件变得非常简单，有了事件，页面就像有了生命一样，活了起来。

2.7 组件通信

本节来讲一讲 React 中的组件通信问题。根据层级关系总共有四种类型的组件通信，分别是父子组件、爷孙组件、兄弟组件和任意组件。需要注意的是，前三种也可以算作任意组件的范畴，所以最后一种是通用方法

2.7.1 父子组件

父子组件间的通信分为父组件向子组件传递消息和子组件向父组件传递消息两种情况。下面先来介绍父组件向子组件传递消息，传统做法分为两种情况，分别是初始化时的参数传递和



实例化阶段的方法调用。

```
class Child {
  constructor(name) {
    // 获取 DOM 引用
    this.$div = document.querySelector('#wp');

    // 初始化时传入 name
    this.updateName(name);
  }
  updateName(name) {
    // 对外提供更新的 API
    this.name = name;

    // 更新 DOW
    this.$div.innerHTML = name;
  }
}
```

```
class Parent {
  constructor() {
    // 初始化阶段
    this.child = new Child('yan');

    setTimeout(() => {
      // 实例化阶段
      this.child.updateName('hou');
    }, 2000);
  }
}
```

在 React 中将两种情况统一处理，全部通过属性来完成。之所以能够这样做，是因为 React 在属性更新时会自动重新渲染子组件。在下面的例子中，2 秒后子组件会自动重新渲染，并获取新的属性值。

```
class Child extends Component {
  render() {
    return <div>{this.props.name}</div>
```




```
    }  
  }  
  
  class Parent extends Component {  
    constructor() {  
      // 初始化阶段  
      this.state = {name: 'yan'};  
  
      setTimeout(() => {  
        // 实例化阶段  
        this.setState({name: 'hou'})  
      }, 2000);  
    }  
    render() {  
      return <Child name={this.state.name} />  
    }  
  }  
}
```

接下来看一下组件如何向父组件传递消息，传统做法也有两种，其中一种是回调函数；另一种是为子组件部署消息接口。

先来看回调函数的例子。回调函数的优点是非常简单；缺点就是必须在初始化时传入，并且不可撤回，且只能传入一个函数。

```
class Child {  
  constructor(cb) {  
    // 调用父组件传入的回调函数，发送消息  
    setTimeout(() => { cb() }, 2000);  
  }  
}  
  
class Parent {  
  constructor() {  
    // 初始化阶段，传入回调函数  
    this.child = new Child(function () {  
      console.log('child update')  
    });  
  }  
}
```



再来看部署消息接口的方法。首先需要可以发布和订阅消息的基类，比如下面实现了一个简单的 `EventEmitter`，在实际生产中会有很多人写好的类库，可直接使用。子组件继承消息基类，就有了发布消息的能力，然后父组件订阅子组件的消息，即可实现子组件向父组件传递消息的功能。

部署消息接口的优点就是可以随处订阅，并且可以多次订阅，还可以取消订阅；缺点是略显麻烦，需要引入消息基类。

// 消息接口，订阅发布模式，类似于绑定事件、触发事件

```
class EventEmitter {
  constructor() {
    this.eventMap = {};
  }
  sub(name, cb) {
    const eventList = this.eventMap[name] = this.eventMap[name] || [];
    eventList.push(cb);
  }
  pub(name, ...data) {
    (this.eventMap[name] || []).forEach(cb => cb(...data));
  }
}
```

```
class Child extends EventEmitter {
  constructor() {
    super();
    // 通过消息接口发布消息
    setTimeout(() => { this.pub('update') }, 2000);
  }
}
```

```
class Parent {
  constructor() {
    // 初始化阶段，传入回调函数
    this.child = new Child();

    // 订阅子组件的消息
  }
}
```



```
    this.child.sub('update', function () {
      console.log('child update')
    });
  }
}
```

Backbone.js 就同时支持回调函数和消息接口方式，但在 React 中选择了比较简单的回调函数模式。下面来看一下 React 的例子。

```
class Child extends Component {
  constructor(props) {
    setTimeout(() => { this.props.cb() }, 2000);
  }
  render() {
    return <div></div>
  }
}

class Parent extends Component {
  render() {
    return <Child cb={() => {console.log('update')}} />
  }
}
```

2.7.2 爷孙组件

父子组件其实可以算是爷孙组件的一个特例，这里的爷孙组件不光指爷爷和孙子组件通信，而是泛指祖先和后代组件通信，可能隔着很多层。我们已经解决了父子组件通信的问题，根据递归思想，很容易得出爷孙组件通信的答案，那就是层层传递属性，把爷孙组件通信分解为多个父子组件通信的问题。

层层传递的优点是非常简单，用已有知识就能解决问题，但是会浪费很多代码，非常烦琐，中间作为桥梁的组件会引入很多不属于自己的属性。

在 React 中，通过 context 可以让祖先组件直接把属性传递给后代组件，有点类似于《星际旅行》中的虫洞，通过 context 这个特殊的桥梁，可以跨越任意层次向后代组件传递消息。

那么如何在需要通信的组件之间开启这个虫洞呢？需要双向声明，也就是在祖先组件中声



明属性，并在后代组件中再次声明属性，然后在祖先组件中放上属性，就可以在后代组件中读取属性了。下面看一个例子。

```
import PropTypes from 'prop-types';

class Child extends Component {
  // 后代组件声明需要读取 context 上的数据
  static contextTypes = {
    text: PropTypes.string
  }
  render() {
    // 通过 this.context 读取 context 上的数据
    return <div>{this.context.text}</div>
  }
}

class Ancestor extends Component {
  // 祖先组件声明需要在 context 上放入数据
  static childContextTypes = {
    text: PropTypes.string
  }
  // 祖先组件往 context 上放入数据
  getChildContext() {
    return {text: 'yanhaijing'}
  }
}
```

context 的优点是省去了层层传递的麻烦，并且通过双向声明控制了数据的可见性，当层数很多时，这不失为一种方案；但缺点也很明显，就像全局变量一样，如果不加以节制很容易造成混乱，而且也容易出现重名覆盖的问题。

个人的建议是对所有组件共享的一些只读信息可以采用 context 来传递，比如登录用户信息等，React Router 路由就是通过 context 来传递路由属性的。

2.7.3 兄弟组件

如果两个组件是兄弟关系，那么可以将父组件作为桥梁来实现两个组件通信，这其实就是



主模块模式。

在下面的例子中，两个子组件通过父组件来实现显示数字同步的功能。

```
class Parent extends Component {
  constructor() {
    this.onChange = function (num) {
      this.setState({num})
    }.bind(this);
  }
  render() {
    return (
      <div>
        <Child1 num={this.state.num} onChange={this.onChange}>
        <Child2 num={this.state.num} onChange={this.onChange}>
      </div>
    );
  }
}
```

主模块模式的优点就是解耦，把两个子组件之间的耦合，解耦成子组件和父组件之间的耦合，把分散的东西收集在一起好处非常明显，能带来更好的可维护性和可扩展性。

2.7.4 任意组件

任意组件包括上面提到的三种关系组件，这三种关系组件之间的通信应该优先使用上面介绍的方法，对于任意两个组件之间的通信，总共有三种方法，分别是利用共同祖先、消息中间件和状态管理。

基于上面介绍的爷孙组件和兄弟组件，只要找到两个组件的共同祖先，就可以将任意组件之间的通信，转化为任意组件和共同祖先之间的通信。这种方法的优点是非常简单，用已知知识就能解决问题；缺点是爷孙组件和兄弟组件这两种模式缺点的叠加，除了临时方案，不建议使用这种方法。

另一种比较常用的方法是利用消息中间件，就是引入一个全局消息工具，两个组件通过这个全局消息工具进行通信。

还记得上面介绍的消息基类吗？在下面的例子中，组件 1 和组件 2 通过全局 event 进行通信。



```
class EventEmitter {
  constructor() {
    this.eventMap = {};
  }
  sub(name, cb) {
    const eventList = this.eventMap[name] = this.eventMap[name] || [];
    eventList.push(cb);
  }
  pub(name, ...data) {
    (this.eventMap[name] || []).forEach(cb => cb(...data));
  }
}

// 全局消息工具
const event = new EventEmitter;

// 一个组件
class Element1 extends Component {
  constructor() {
    // 订阅消息
    event.sub('element2update', () => {console.log('element2 update')});
  }
}

// 另一个组件
class Element2 extends Component {
  constructor() {
    // 发布消息
    setTimeout(function () { event.pub('element2update') }, 2000)
  }
}
```

消息中间件利用观察者模式,将两个组件之间的耦合解耦成组件和消息中心+消息名称的耦合。但为了解耦,却引入了全局消息中心和消息名称,消息中心对组件的侵入性很强,和第三方组件通信不能使用这种方式。

对于小型项目,比较适合使用这种方式,但随着项目规模的扩大,达到中等规模以后,消

息名称呈爆炸式增长，对消息名称的维护成了棘手的问题，重名概率极大，没有人敢随便删除消息信息，消息发布者找不到消息订阅者的信息等。

其实上面的问题也不是没有解决办法，重名的问题可以通过制定规范、消息命名空间等方法来极大地降低冲突，其他问题可以通过把消息名称统一维护到一个文件中，通过对消息的中心化管理来解决。

如果项目规模非常大，上面两种方案都不合适，那么可能需要一个状态管理工具，通过状态管理工具把组件之间的关系和关系的处理逻辑从组件中抽象出来，并集中化到统一的地方来处理。Redux 就是一个状态管理工具，本书后面会详细介绍 Redux，这里不再赘述。

总体来说，虽然组件间的关系千变万化，但是都可以用上面介绍的方法来解决问题，对于不同规模的项目，应该选择适合自己的技术方案。

2.8 组件的抽象与复用

如果一段代码在两个地方出现，那么应该复用，在 React 中可以分为组件级别的复用和逻辑代码段级别的复用。关于如何设计公共组件、如何抽象复用，下面介绍一些方法和经验。

2.8.1 组件的设计与抽象

在抽象公共功能时，要尽可能通用、灵活，高内聚、低耦合，对于 React 组件可以总结出下面一些经验和规则。

- 组件应该只通过属性输入，避免通过 context，更要避免读取全局变量、系统 I/O 等。
- 组件的属性应该有默认值，这样使用起来更简单，在大多数情况下不需要传递很多参数。
- 组件的属性应该使用简单值，尽量避免使用对象等复杂的数据结构，简单的属性值更容易理解和维护。
- 组件要足够健壮，考虑边界异常情况，要做好属性的类型验证，不可缺省。
- 组件要有灵活的适用能力，不要限制使用环境，而要适用于一切环境，比如组件不要给自己设置宽度，要适用于所有的宽度。

2.8.2 继承

如果两个组件有一部分功能是一样的,那么可以通过抽象一个父类,然后通过继承的方式,解决重复的问题。

比如很多组件都使用 `shouldComponentUpdate` 实现浅比较来提升性能,多个组件单独定义,就存在重复的问题。

```
// shallowCompare 的定义此处忽略
class Demo1 extends Component {
  shouldComponentUpdate(nextProps, nextState) {
    const { props, state } = this;
    return shallowCompare(nextProps, props) && shallowCompare(nextState, state);
  }
}
```

```
class Demo2 extends Component {
  shouldComponentUpdate(nextProps, nextState) {
    const { props, state } = this;
    return shallowCompare(nextProps, props) && shallowCompare(nextState, state);
  }
}
```

可以抽象一个父类组件 `PureComponent`,把重复的部分提取到父类中来解决这个问题。

```
class PureComponent extends Component {
  shouldComponentUpdate(nextProps, nextState) {
    const { props, state } = this;
    return shallowCompare(nextProps, props) && shallowCompare(nextState, state);
  }
}
```

```
class Demo1 extends PureComponent {
  // 继承父类的 shouldComponentUpdate
}
```

```
class Demo2 extends PureComponent {
  // 继承父类的 shouldComponentUpdate
}
```

继承强调的是 A is B (A 是 B)。也就是说，子类必须是父类，子类是父类的一个更狭隘的定义。比如卡车的父类是汽车，但汽车不是自行车的父类。这个例子非常简单，但是有时候父类不是很容易确定，比如卡车和人的父类就不太容易确定。

如果继承设计得不好，到后面会变得非常脆弱、不好维护，所以关于继承一定要谨慎，如果想不清楚就不要抽象父类。

2.8.3 组合

不是所有的重复问题都能通过继承来解决，继承必须是 A is B 的关系，对于 A has B 的关系，可以通过组合来实现复用。

比如汽车和人都会跑，人和跑不是“是”的关系，而是“拥有”的关系，人有跑的功能，如果用继承的方法，这个父类就很难抽象，人和汽车都继承跑吗？显然不是很合理，然而组合可以很好地解决问题。

在 Java 中实现组合的方式就是接口继承，在 JavaScript 中实现组合的方式有很多种，下面进行简单介绍。

在上面汽车和人的例子中，重复的部分是跑，先把跑这个功能抽象出来，比如抽象出“跑”和“停”两个方法。

```
const map = {  
  run() {  
    this.runState = true;  
  }  
  stop() {  
    this.runState = false;  
  }  
};
```

最简单的组合方式就是内部调用，手工提供一层代理，需要处理 this 的指向和参数的传递问题。

这种方式最容易理解，但手工方式还是有一部分重复的，而且如果复用的函数很多的话，则需要写很多代码。

```
class People {  
  run(...args) {
```

```

    return map.run.call(this, ...args);
  }
  stop(...args) {
    return map.stop.call(this, ...args);
  }
}

```

更好一点的做法是批量拷贝，可以自己实现一个 `extend` 函数，也可以借用 `jQuery` 里的 `$.extends` 方法，或者 ES 6 引入的 `Object.assign` 方法。

这种方式存在的问题在于，组合是一次性的，因为拷贝是只复制一次值，如果后续 `map` 中的 `run` 方法被重新赋值，新的值并不会再次被自动拷贝。

```

function extend(obj1, obj2) {
  Object.keys(obj2).forEach(function (key) {
    obj1[key] = obj2[key]
  })
}

class People {}

```

```

// 下面三行代码的功能一样，都可实现组合
extends(People.prototype, map);
$.extends(People.prototype, map);
Object.assign(People.prototype, map);

```

在非 ES 6 语法的 `React` 中，原生提供了 `mixin` 功能。比如每个设置定时器的组件都需要在组件卸载时清除定时器，就可以把这个功能抽象出来。

```

var SetIntervalMixin = {
  componentWillMount: function() {
    this.intervals = [];
  },
  setInterval: function() {
    this.intervals.push(setInterval.apply(null, arguments));
  },
  componentWillUnmount: function() {
    this.intervals.forEach(clearInterval);
  }
};

```



```
const Demo1 = React.createClass({
  mixins: [SetIntervalMixin]
});
```

```
const Demo2 = React.createClass({
  mixins: [SetIntervalMixin]
});
```

2.8.4 高阶组件

在 ES 6 语法的 React 中，并没有原生提供 mixin 功能，但通过上面提到的 extends 也可以实现类似的功能。

```
var SetIntervalMixin = {
  // 省略
};

class Demo1 extends Components {

}

extends(Demo1.prototype, SetIntervalMixin);
```

但 mixin 方法存在重名覆盖的问题，就是后面混入的重名方法会覆盖前面混入的方法，当多人维护项目时，或者大量引用第三方 mixin 时，这个问题会被放大。

React 社区给出的另一种解决办法是使用高阶组件。高阶组件就是接收组件，并且返回组件的组件。实现高阶组件有两种方法，一种是调用传入的组件；一种是继承传入的组件。

```
// 调用传入的组件
function HOC1(InnerComponent) {
  return class WrapComponent extends Components {
    render() {
      return (
        <InnerComponent ...this.props>
          {this.props.children}
        </InnerComponent>
      );
    }
  };
}
```

```

    });
  }
}

let Demo1 = class extends Components {

}

Demo1 = HOC1(Demo1);

// 继承传入的组件
function HOC2(InnerComponent) {
  return class WrapComponent extends InnerComponent {}
}

let Demo2 = class extends Components {

}

Demo2 = HOC1(Demo2);

```

一般只在传入组件的外围进行一些操作时使用第一种方法；如果想在传入组件的内部进行一些操作，比如改写 `render`，则使用第二种方法。

通过 `super` 关键字，可以解决 `mixin` 重名覆盖的问题，可多次定义重名方法，比如前面提到的清除定时器的例子，用高阶组件实现如下：

```

function SetTimeoutHOC(InnerComponent) {
  return class WrapComponent extends InnerComponent {
    componentWillMount() {
      super.componentWillMount();
      this.timeouts = [];
    },
    setTimeout() {
      super.componentWillMount();
      this.timeouts.push(setTimeout.apply(null, arguments));
    },
    componentWillUnmount() {
      super.componentWillMount();
      this.timeouts.forEach(clearInterval);
    }
  }
}

```

```
    }  
  }  
  
  function SetIntervalHOC (InnerComponent) {  
    return class WrapComponent extends InnerComponent {  
      componentWillMount() {  
        super.componentWillMount();  
        this.intervals = [];  
      },  
      setInterval() {  
        super.componentWillMount();  
        this.intervals.push(setInterval.apply(null, arguments));  
      },  
      componentWillUnmount() {  
        super.componentWillMount();  
        this.intervals.forEach(clearInterval);  
      }  
    }  
  }  
}  
  
let Demo1 = class extends Components {  
  
}  
  
Demo1 = SetTimeoutHOC (SetIntervalHOC (Demo1));
```

总结：继承、组合和高阶组件通过三种不同的思路，都可以解决代码重复的问题，在实际工作中需要活学活用，根据不同的场景分别使用不同的方法。

2.9 命令式与 DOM

在 React 之前，前端是基于命令式和 DOM 来编程的。而 React 把 UI 抽象为组件和状态，在 React 的世界里，在大多数情况下不需要命令式和 DOM，但在有些情况下 React 的规则并不适用。本节我们就来学习命令式的 React 和 React 中的原生 DOM 操作。

2.9.1 ref

React 通过 ref 给了我们引用组件和 DOM 元素的能力，这在某些情况下会非常有用。

如果一个表单元素的值不存在 state 中，也就是没有绑定 onChange 方法，这时想获取所输入的值，则可以通过 ref 来实现。通过 ref 也可以调用原生 DOM 的方法，比如表单元素获取焦点。

```
class User extends Component {
  constructor() {
    setTimeout(() => {
      // 通过 ref 可以获取输入框的值
      console.log(this.nameInput.value);

      // 获取焦点
      this.nameInput.focus();
    }, 2000);
  }
  render() {
    return (
      // ref 的值是一个函数，会在 componentDidMount 和 componentDidUpdate 后执行
      <input ref={(input) => this.nameInput = input} type="text">
    );
  }
}
```

以上让输入框获取焦点的方法，就是典型的命令式编程，和 React 操作 state 的思路完全不同。

通过 ref 同样可以获取 React 组件，并调用组件中的方法。一个典型的例子就是弹窗组件，很多弹窗组件都是通过命令式来打开和关闭的，但更符合 React 的做法应该是通过一个 isOpen 属性来控制。

```
class Dialog extends Component {
  alert() {
    this.setState({status: 1})
  }
  close() {
```

```
        this.setState({status: 0})
    }
}

class Home extends Component {
  constructor() {
    // 打开弹窗
    setTimeout(() => {this.dialog.alert()}, 1000);
    // 关闭弹窗
    setTimeout(() => {this.dialog.close()}, 8000);
  }
  render() {
    return (
      <Dialog ref={(dialog) => {this.dialog = dialog}}></Dialog>
    )
  }
}
```

另一个可能用到原生 DOM 的场景就是使用了非 React 的第三方库，这可以弱化 React 最初生态不足的缺点。在下面的例子中使用了 jQuery 的 autocomplete 组件。

```
class User extends Component {
  componentDidMount() {
    // 获取 DOM，复用已有的第三方库
    $(this.textInput).autocomplete();
  }
  render() {
    return <input type="text" ref={(input) => this.textInput = input}>
  }
}
```

2.9.2 findDOMNode

使用 findDOMNode 可以获取整个组件的 DOM，但有了 ref，能用到 findDOMNode 的地方就非常少了，一个常见的例子是在父组件中获取子组件的完整 DOM。

```
class Parent extends Component {
  componentDidMount() {
```



```
// 获取子组件的完整 DOM
ReactDOM.findDOMNode(this.child);
}
render() {
  return <Child ref={(child) => this.child = child}></Child>
}
}
```

2.9.3 dangerouslySetInnerHTML

React 对输出的内容都会进行 XSS 过滤，但是在某些情况下却不要这个功能，比如在接口返回 HTML 片段的情况下，`dangerouslySetInnerHTML` 可以将 HTML 片段直接设置到 DOM 上。

```
function User() {
  return <div dangerouslySetInnerHTML={__html: '<a>yanhaijing.com</a>'}></div>
}
```

对于刚刚接触 React 的读者，建议忘记命令式和原生 DOM，先熟悉 React 的规则，只有这样才能在合适的场景中使用合适的方法，达到事半功倍的效果。切记：不要滥用本节介绍的内容。

2.10 本章小结

本章围绕组件介绍了很多 React 相关知识，包括组件语法、JSX、生命周期、属性和状态、绑定事件、组件间通信、原生 DOM 等，学会这些内容已经可以满足日常开发需求了。但还有很多知识没有介绍，比如测试、底层实现、性能优化、动画相关知识等，希望读者能够自己了解相关内容，随着了解的深入，以及业务的演变，相信一定会掌握这些知识点。

另外，React 本身也在快速演进，本章介绍的 API 可能会被废弃，本章介绍的方法可能会显得陈旧，希望大家能够学到本质，并保持持续学习。

React 并不适合用来写 Web 页面，React 适用的场景是富交互的大型网站，或者 WebApp，一定要根据具体的产品场景选择合适的技术。

React 只是招式，JavaScript 语言才是内功，切勿只学其一，内外兼修方能发挥最大价值。

第 3 章

Redux 应用架构基础

作为前端开发者，相信即使没有使用 React 开发项目的经验，也一定听说过 Redux 的大名；如果你只是单纯的 React 初级使用者，那么一定对 Redux 充满好奇、跃跃欲试。

没错，作为 Redux 的作者，Dan Abramov 本人也没有想到，Redux 自 2015 年横空出世后，很快席卷了 React 社区，并备受推崇。那么它究竟是什么，怎么使用呢，又能解决 React 传统开发中的哪些痛点呢？

本章将会为大家介绍 Redux 基础及用法。笔者认为，从头开始系统学习一个框架或类库的最好方式一定是阅读官方文档。对于本章内容，建议配合官方文档进行学习。Redux 在一定程度上增加了 React 技术栈的学习成本，但是作为开发者，一定不要望而却步，其实它是一个非常有趣而简单的“魔法”，相信通过阅读本章内容，你会很快理解并掌握 Redux。

3.1 Redux 究竟是什么

通过前面章节的学习，我们了解到 React 应用离不开组件，而组件的状态 (state) 非常关键，它往往是一个或多个组件的核心。打一个形象的比方：对于一个成型的产品，若所有组件构成了其骨架，那么状态就是贯穿骨架各个关节的血液。

我们同时了解到，随着应用越来越复杂、用户交互越来越丰富，不同组件之间的通信也会变得越来越必要且频繁。React 框架提倡的单向数据流理念，在一定程度上将复杂的数据通信变得简单，这是区别于操作 DOM 型类库（例如 jQuery）和传统 MVC 型类库的显著优点。

可是，这往往并不能完全有效地解决问题。比如，我们还需要通过 `setState` 方法细粒度地控制状态的变化，并设计组件之间状态的通信。长此以往，状态很有可能在团队不同的开发者手

下“野蛮增长”而变得混乱，状态的变化因为无迹可寻而不可预测，即使对于同一个开发人员来说，状态也会变得不易掌控。

而此时 Redux 作为一个应用数据流架构，和 React 完美结合，能够在一定程度上解决上述问题。

3.1.1 Redux 是库结合模式

了解了背景后，我们来看看 Redux 究竟是什么。

Redux 的官方定义是：Redux is a predictable state container for JavaScript apps.

翻译过来就是，对于 JavaScript 应用而言，Redux 是一个可预测状态的“容器”。不过初次接触 Redux 的开发者，也许并不能从这样简单的一句描述中领会太多信息，对 Redux 的认识仍然停留在一个抽象的概念当中。

现在用我们熟悉的概念对 Redux 进行解读。Redux 是一个库吗？——是的，它是一个体积很小且优雅的库。但是相比我们熟悉的 jQuery 甚至 Underscore 这种库来说，Redux 又不仅仅是一个库。它约定了开发者在使用时的条条框框：“做这个，做那个”“不能这样，你要那样”等。在这一层面上，它又像是一种设计模式。因此，不妨将 Redux 理解成一个严格规定了使用模式的库。

事实上，Redux 借鉴了函数式编程的思想，采用了单向数据流理念，实现起来非常优雅。配合 React，它既不操作 DOM，也不是实践 MVC 的重型武器，它只专注于全局状态的管理，为实现对数据状态的管理封装提供了不同的方法，并规定了我们进行管理的模式。

因此，它通过这一系列的“魔法”和“约定模式”，使得数据状态变得可预测、可追溯。这种可预测性和可追溯性，对我们的编程体验和代码维护，以及 Bug 排查是极其重要的。

这样说仍然有些抽象，请读者继续阅读以下章节之后再回来体会“Redux 究竟是什么”，相信一定会有所领悟。同时，对于所谓的可预测性、可追溯性，初学者也许难以马上理解它们的含义。但不要急，相信真正上手开发项目或阅读更多内容后，一定能够体会其中的奥秘。

3.1.2 Redux 和 React 的关系

接下来，我们讨论一下 Redux 和 React 是什么关系。

其实它们没有任何关系, Redux 同样可以与 jQuery 等类库搭配使用。但是就如前面所说的, React 应用规模一旦变大, 就需要对状态格外关注和维护, 而此时 Redux 能够“规范化”状态的改变。因此, 两者能够产生不可思议的“化学反应”。这也是它们如同孪生兄弟一样总是结伴出现的原因。但是这一对孪生兄弟从生物学角度来讲, 是毫无血缘关系的。请进入后面章节, 我们一探真相。

3.2 Redux 设计哲学

在具体了解 Redux 的使用之前, 我们有必要了解一下 Redux 的设计哲学。首先回到 React 上, 读者已经理解: 使用 React 时, 应用就是一个个状态机, 页面中的组件在得到所需状态之后, React 便会生成一个视图 (View), 在状态机的状态发生改变后, 又会由 React 生成一个对应状态的视图。

由此, 我们来尝试理解 Redux 的第一个哲学理念: Single source of truth.

简单翻译成“单一真相”可能会令开发者一头雾水。事实上, 这里表达的是数据来源的单一。

不论是页面应用像一个计数器一样简单, 还是像一个聊天系统般复杂, 我们都使用一个 JavaScript 对象来表述整个状态机的全部状态, 并存储在 store 当中, 这个 store 一定是唯一的, 它就是 React 组件依赖的“Single source of truth”。

换句话说, 页面状态数据树被存储为一个 JavaScript 对象, 这棵页面状态数据树是会随着用户的操作或者异步数据的到达等变化而发生变更的。任何时刻我们都可以使用下面方式来获取当前应用的数据状态。

```
let state = store.getState();
```

这样的设计理念有什么好处呢?

最重要的是, 这个哲学理念让我们的关注点变得非常直接和简单。因为页面的展现内容依赖各个组件的状态数据, 组件的状态数据又存储在一个 JavaScript 对象当中, 而这个对象可以随时可以通过 store.getState 方法获取, 因此, 我们只需要把注意力完全集中在这个对象 (即 store.getState 返回值) 上即可。这样一来, 开发者的精力便得到聚焦, 排除了其他环节对开发和维护的干扰。

接下来, 我们来看一下 Redux 的第二个哲学理念: State is read-only.

前面提到的页面状态数据树，即 `store.getState` 返回的结果是只读的，只读就意味着我们不能直接改变它。类似下面的做法是不被允许的：

```
let state = store.getState();
state.foo = 'bar';
```

问题来了，这个存储着页面数据的 JavaScript 对象是只读的，可是页面不能只有一棵一成不变的状态数据树，不然还怎么做出响应并更新页面呢？

我们姑且先不考虑如何让一个 JavaScript 对象只读，因为这里的“只读”并不是“保护一个对象不受改变”，而是当页面需要新的数据状态时再生成一棵全新的状态数据树，使得 `store.getState` 返回一个全新的 JavaScript 对象。

那么一定有一个类似于 `store.makeNewState` 的方法吧？其实，Redux 对生成一个全新的页面状态数据对象进行了拆解，它规定：当页面需要展现新的数据状态时，我们只需要 `dispatch`（派发）一个 `action`（动作/事件）即可。这个 `action` 其实也是一个 JavaScript 对象，就像页面状态数据树这个 JavaScript 对象描述了整个页面的状态一样，`action` 则描述了这个动作单元变化的所有信息。

这时，你可能会有疑问：“派发相应的 `action` 用来描述页面状态数据树需要做出的改变，那么这个改变到底是怎么执行，进而产生新的页面数据状态的呢？”

这就不得不提 Redux 的第三个哲学理念：Changes are made with pure functions called reducer.

使用 `reducer` 函数来接收 `action`，并执行页面状态数据树的变更。经过 `reducer` 函数处理之后，`store.getState` 方法就会返回新页面的数据状态。

当然，前面提到的页面状态数据树对象是只读的，经过 `reducer` 函数处理之后，返回了一个新的 JavaScript 对象，而不会对原返回值进行更改。

所以，`reducer` 并不直接更改页面状态数据树：

```
function reducer() {
  let state = store.getState();
  state.fpp = 'bar';
}
```

而是根据当前的页面状态数据树，结合描述改变信息的 `action`，产生一棵新的页面状态数据树，并把它应用在 `store.getState` 当中。

reducer 和 action 都需要由开发者编写。reducer 接收以下两个参数：

- 当前页面数据状态。
- 被派发的 action。

所以这个函数的处理可以抽象表达出来：

```
(previousState, action) => newState
```

这样一切变得清晰而简单。

细心的读者可能会考虑 reducer 函数的命名由来。因为在 ES 5 中，数组存在方法 `Array.prototype.reduce` 中，这是一种巧合吗？事实上，在 Redux 源码 GitHub 仓库中也有官方解释：

“It’s called a reducer because it’s the type of function you would pass to `Array.prototype.reduce(reducer, ?initialValue)`.”

虽然是一笔带过的，但是总结得恰到好处。JavaScript 数组的 `reduce` 方法是一种运算合成，它通过遍历、变形、累积，将数组的所有成员“累积”为一个值。MDN 中的描述则更加直接：对累加器和数组中的每个值（从左到右）应用一个函数，以将其减少为单个值。

在 Redux 数据流里，reducer 在具备初始状态的情况下，每一次运算其实都是根据之前的状态（previous state）和现有的 action（current action）来更新 state 的，这个 state 可以理解为上文中累加器的结果。每次 reducer 被执行时，state 和 action 都被传入，这个 state 根据 action 进行累加，进而返回最新的 state。这符合一个典型 reduce 函数的用法和思想，其实这也是函数式编程的一个重要概念和体现。

下面我们结合图示进行总结。Redux 追求单一数据源，所有的状态数据都存储在 store 当中。store 和 state 的关系如图 3-1 所示。



图3-1 store和state关系图

状态数据是只读的，当页面需要做出改变时，这个改变由 action 描述，并被开发者派发。action 触发状态更新示意图如图 3-2 所示。

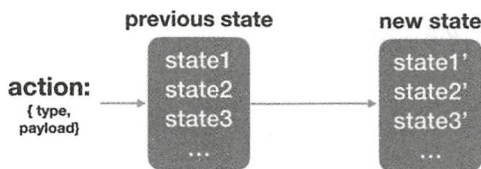


图3-2 action触发状态更新示意图

这个 action 被相应的 reducer 作为参数获取, 并根据现有的页面状态数据树产生一棵新的页面状态数据树。reducer 角色如图 3-3 所示。

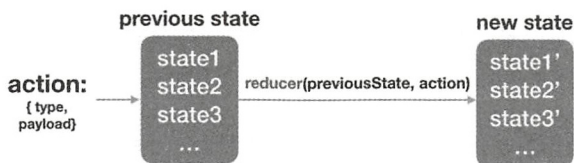


图3-3 reducer角色图

再来结合 React, 当页面中各个 React 组件根据新的页面状态数据树抽取出自己需要的状态进行更新, 并触发重新渲染之后, 就得到了我们预期的页面。Redux 整体流程图如图 3-4 所示。

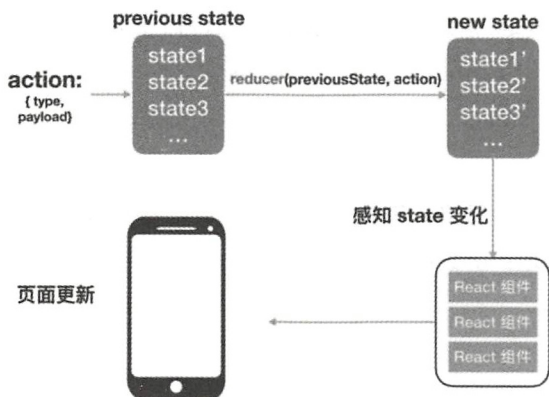


图3-4 Redux整体流程图

通过本节介绍, 希望读者对 Redux 的设计哲学能有一个大致了解, 一些抽象的概念能够在流程上落地。当然, 对于很多细节, 读者可能仍有疑问, 比如 dispatch 是什么, 如何 dispatch 一个 action 等。请读者继续阅读。

3.3 函数式编程和纯函数

在具体介绍 Redux 的使用方法和细节之前，先讲解一个非常重要的概念——函数式编程。一些前端开发者认为 Redux 技术栈之所以学习成本较高，在很大程度上和 React 及 Redux 的函数式编程（functional programming）理念有关。另一方面，由于 React 结合 Redux 技术的流行，前端社区中的函数式编程浪潮也可谓“忽如一夜春风来”，持续发酵成为热点。

函数式编程是一种编程范型。编程范型是指一种编程风格，除函数式编程以外，还有大家都很熟悉的面向对象编程等。正如武侠世界里存在众多门派一样，在软件工程中也存在多种多样的编程范型。函数式编程是一种典型的声明式编程，与命令式编程相对立，它更看重程序的执行目标而不是执行过程。

事实上，函数式编程是一个很“古老”的编程理念，这里不会再解释深究。但是与 Redux 相关的内容，以及 Redux 体现函数式编程的地方，我们一定要理解清楚。

3.3.1 了解“函数是一等公民”

函数式编程的一大特点是，函数是“一等公民”。下面我们通过具体实例来对该特点进行阐释。假如需要完成 $(1+2) \times 3 - 4$ 的运算，传统的面向过程式为：

```
let a = 1 + 2;
let b = a * 3;
let c = b - 4;
```

而函数是“一等公民”的思路，则意味着函数优先，提倡使用函数组合。事实上，熟悉 JavaScript 的开发者已经知道：函数可以赋值给其他变量，也可以作为另一个函数的参数，还可以作为别的函数的返回值。

我们先实现以下函数：

```
const add = (x, y) => x + y;
const multiply = (x, y) => x * y;
const subtract = (x, y) => x - y;
```

上述求值过程在函数式编程的理念下，就可以通过连接函数得到：

```
let result = subtract(multiply(add(1,2), 3), 4);
```

我们看到，函数可以与其他数据一样，作为参数传递，或作为返回值返回，这就是函数是

“一等公民”的体现。

3.3.2 了解纯函数和副作用

通过前面的介绍，我们已经清楚了页面状态数据树的只读性，`reducer` 函数的处理便是根据 `action` 参数产生一棵新的页面状态数据树。这其实“暗合”（实际上是主动接受的）了函数式编程当中纯函数的概念。

纯函数代表这样一类函数：

- 对于指定输出，返回指定结果。
- 不存在副作用。

也就是说，纯函数的返回值只依赖其参数。比如：

```
// 这是一个纯函数
const addByOne = x => x + 1;
```

同时，在纯函数内不能存在任何副作用，包括但不限于：

- 调用系统 I/O 的 API、`Date.now()` 或者 `Math.random()` 等方法。
- 发送网络请求。
- 在函数体内修改外部变量的值。
- 使用 `console.log()` 输出信息。
- 调用存在副作用的函数等。

因为这些操作都具有不确定性，是“不纯”的。换句话说，对于纯函数，如果是同样的参数，则一定能得到一致的返回结果。作为开发者，根据其输入，是完全可以预测输出的。

同样，纯函数也不允许内部改变参数值。比如 `addByOne` 函数需要对一个整型数组的每一项进行加 1 操作，为了保持纯函数的特性，我们需要进行如下操作：

```
// 这是一个纯函数
const addItemOneByOne = array => array.map(addByOne);
```

这样通过 `Array.prototype.map` 方法就可保证返回新的数组，而不会影响输入参数。

```
let array1 = [1, 1];
```

```
let array2 = addItemOneByOne(array1);
console.log(array1); // [1, 1];
console.log(array2); // [2, 2];
```

反过来看，像下面这样的操作，明显是有悖于纯函数的特性的，因此它不是一个纯函数。

```
// 这不是一个纯函数
const addItemOneByOne = array => {
  let arrayLength = array.length;
  for (let i = 0; i < arrayLength; i++) {
    array[i] = addByOne(array[i]);
  }
}

let array3 = [1, 1];
let array4 = addItemOneByOne(array3);
console.log(array3); // [2, 3];
console.log(array4); // [2, 2];
```

在了解了这些概念之后，便可知道：Redux 中所有由开发者定义的 reducer 函数，都需要是纯函数。我们再来回顾一下：reducer 接收当前页面状态数据树和 action，在不改变参数页面状态数据树的原则下，返回一棵新的页面状态数据树，并且这棵新的页面状态数据树完全可以通过旧的参数页面状态数据树推测得到的。

为什么会有这样的限制和设计呢？或者说，为什么 Redux 会遵循函数式编程的理念，产生这样的要求呢？

我们想象一个很现实的场景：假如发现程序中存在一个 Bug，页面需要展示数据 1，但是错误地展示了数据 2。在 Redux 架构下，整个修复思路就变得非常清晰：出现任何问题一定是因为组件接收了不正确的 state，那么这个 state 的产生出自 reducer。我们先来看产生这个错误 state 的 reducer 接收到的 action 内容是否正确，如果正确，则说明 action 准确表达了需要做出的改变，那么很有可能就是 reducer 函数的内部处理出现了错误。

这就是之前提到的可预测性、可追溯性的体现——使得调试、维护代码非常简单。其实 Redux 涉及的函数式编程并不止这些，纯函数还会牵扯出不可变性和共享数据的概念，在 3.6 节会深入介绍。

3.4 Redux 基本使用和实践

讲到这里，也许读者已经大体掌握了 Redux 的流程和架构，但是由于不了解其 API 及具体实践方法，多少有些“镜花水月”之感。本节我们就从实践出发，让读者对 Redux 有更深入的理解。

3.4.1 初识 store

我们已经不止一次提到 store，store 是 Redux 中最核心的概念，是 Redux 架构的根本。前文中提到过 Redux 是一个可预测状态的“容器”，这里所说的容器，其实就是指 store。这个容器体现了前面提到的 Redux 的三个哲学理念，保存着整个页面状态数据树，并且为开发者提供了重要的 API。

事实上，store 就是一个 JavaScript 对象，里面包含了 dispatch 及获取页面状态数据树的方法等。

```
store = {  
  dispatch,  
  getState,  
  subscribe,  
  replaceReducer  
}
```

我们来看看 store 本身提供给开发者使用的方法有哪些。

- dispatch(action): 派发 action。
- subscribe(listener): 订阅页面数据状态，即 store 中 state 的变化。
- getState: 获取当前页面状态数据树，即 store 中的 state。
- replaceReducer(nextReducer): 一般开发用不到，社区一些热更新或者代码分离技术中可能会使用到。

那么如何创建一个 store 呢？当引入 Redux 之后，我们便可以使用 Redux.createStore 方法来创建页面应用的 store，即产生一个对象实例：

```
import { createStore } from 'redux';  
const store = createStore(reducer, preloadedState, enhancer);
```


`createStore` 方法可以接收三个参数。

- **reducer**: 为开发者编写的 reducer 函数，必需。
- **preloadedState**: 页面状态数据树的初始状态，可选。
- **enhancer**: 增强器，函数类型，可选。

其中 **reducer** 参数必须存在。也就是说，当开发者创建一个 **store** 时，必须同时定义好 **reducer** 函数，用来告知 **store** 数据状态如何根据 **action** 进行变更。

在正常开发中，我们只需要关心前两个参数来创建 **store** 就足够了。在 **Redux** 应用架构中，**store** 必须是唯一的。这也是 **Redux** 区别于 **Flux** 等架构的一个显著特点。

关于 **store** 的 **replaceReducer** 方法，以及创建 **store** 时的 **enhancer** 参数，它们都属于高级用法，更深入的介绍请参考后续章节。

3.4.2 构造 action

action 描述了状态变更的信息，也就是需要页面做出的变化。这是由开发者定义并借助于 **store.dispatch** 派发的。**action** 本质上也是一个 **JavaScript** 对象。为了清楚和统一，**Redux** 规定 **action** 对象需要有一个 **type** 属性，作为描述这个 **action** 的名称来唯一确定这个 **action**，一般我们采用 **JavaScript String** 类型。另外，**action** 往往还需要携带一些数据信息，这些数据信息的属性名没有要求，同时这些数据信息中包含了这个 **action** 变化的基本内容。

比如，如下对象就是一个标准的 **action**，它描述了一种变化，并传递数据 **action1.data** 给 **store** 中的 **reducer** 处理。这种变化可以理解为“阅读 **Redux** 书籍”，即 **action1.type**，那么具体阅读哪一本书呢？就是 **action1.data** 所携带的 **action1.data.book**：“深入浅出 1”。

```
const action1 = {  
  type: 'READ_REdux_BOOK',  
  data: {  
    book: '深入浅出 1'  
  }  
}
```


3.4.3 使用 action creator (构造器)

想象这样的情况：我们需要一些 type 为 READ_REDUX_BOOK 的 action，用来描述阅读 Redux 书籍这样一类变化。但是所阅读的书籍每本都不同，即这些 action 每次携带的数据都不同。比如：

```
const action2 = {
  type: 'READ_REDUX_BOOK',
  data: {
    book: '深入浅出 2'
  }
}

const action3 = {
  type: 'READ_REDUX_BOOK',
  data: {
    book: '深入浅出 3'
  }
}

const action4 = {
  type: 'READ_REDUX_BOOK',
  data: {
    book: '深入浅出 4'
  }
}
```

这种情况在开发中非常多见。想象一下：action 携带的数据来自用户输入，每个用户都会推荐不同的 data.book，那么我们可以定义这样一个 action creator 函数：

```
const learnReduxActionFactory = book => {
  type: 'READ_REDUX_BOOK',
  book
}
```

这个函数返回一个对象，即一个 action，但是 action.type 固定为 READ_REDUX_BOOK，而其所携带的信息由参数决定。这就类似于一个工厂模式的生产工具，也算是一种非常常见的“最佳实践”。

3.4.4 使用 dispatch 派发 action

介绍到这里，store.dispatch 函数已经不再神秘——dispatch 来自 store 对象暴露的方法，负责派发 action，这个 action 将作为 dispatch 方法的参数。以上面定义的 action 为例：

```
store.dispatch(action1);
```

在使用 action creator 的情况下，便有：

```
store.dispatch(learnReduxActionFactory('深入浅出1'))
```

3.4.5 编写 reducer 函数更新数据

action 描述了一种变化，并携带这种变化的数据信息。真正执行这种变化并生成正确数据状态的是 reducer 方法。我们了解到，reducer 必须为纯函数，以保证数据变化的可预测性。下面的 updateStateTree 就是一个 reducer 函数。

```
const updateStateTree = function (previousState = {}, action) {  
  // ...  
  return newState;  
}
```

当然，一个完整的 reducer 函数可能需要对多个 action 进行处理，所以在开发时往往使用 switch-case 或者 if-else 来编写，代码如下。

```
const updateStateTree = function (previousState = {}, action) {  
  switch (action.type) {  
    case 'case1':  
      return newState1;  
    case 'case2':  
      return newState2;  
    default:  
      return previousState  
  }  
}
```

注意：当无法匹配 action 时，默认返回原 previousState 参数，以保证其返回值的稳定性。

通常在页面初加挂载时，我们就需要定义一个初始页面状态数据树来展示默认状态。注意，updateStateTree 函数的第一个参数 previousState = {} 表示设置了默认值代表初始状态。在代码中

这个默认值是一个空对象，当然开发者可以根据需要合理进行设置。

总结：当通过 Redux 的 `createStore` 方法创建了一个 `store` 实例之后，我们便可以使用 `store.dispatch` 派发一个描述变化的 `action`，这个 `action` 需要开发者结合自身业务进行编写。同时，在执行 `store.dispatch` 之后，Redux 会“自动”帮我们执行处理变化并更新数据的 `reducer` 函数。从 `store.dispatch` 到 `reducer` 这个过程可以认为是由 Redux 内部处理的，但是具体的 `action` 及 `reducer` 需要开发者编写，以完成应用开发需求。那么当页面数据状态得以更新之后，如何促使页面发生 UI 更新呢？实际上就需要使用 `store.subscribe(callbackFunction)` 方法订阅数据的更新，并由 `callbackFunction` 完成 UI 更新。

3.4.6 合理拆分 reducer 函数

当业务变得复杂，需要由几个或者几十个甚至更多的 `action` 来描述不同的变化时，`reducer` 函数也将变得庞大无比，在函数内部可能就要针对不同的 `action` 进行不同的逻辑处理。例如：

```
const fatReducer = function (previousState = {}, action) {  
  switch (action.type) {  
    case 'case1':  
      // 计算逻辑  
      return newState1;  
    case 'case2':  
      // 计算逻辑  
      return newState2;  
    .....  
    .....  
    case 'caseN':  
      // 计算逻辑  
      return newStateN;  
    default:  
      return previousState  
  }  
}
```

这样一个庞大的函数，显然不能算是“最佳实践”，这对于开发体验和维护成本都将是难以接受的。

为了解决这个问题，Redux 提供了一个工具函数：`combineReducers`，借助于它我们可以对 `reducer` 函数进行拆分，最后再合并为一个完整的 `reducer`。它接收一个 JavaScript 对象类型的参数，这个对

象的键值分别为页面数据状态分片和子 reducer 函数，最后返回一个合并归一的 finalReducer。

```
let finalReducer = combineReducers({reducers});
```

比如，在页面数据状态中存在三种数据状态：data1、data2、data3，它们相互独立而不关联。

```
state = {  
  data1: {  
    ...  
  },  
  data2: {  
    ...  
  },  
  data3: {  
    ...  
  }  
}
```

我们把这三种数据状态拆分成三个小的 reducer 函数进行处理：reducer1、reducer2 和 reducer3。

```
const reducer1 = function (previousState = {}, action) {  
  // 根据 action 和 state.data1 计算产生新的 state.data1  
  return state.data1;  
};  
const reducer2 = function (previousState = {}, action) {  
  // 根据 action 和 state.data2 计算产生新的 state.data2  
  return state.data2;  
};  
const reducer3 = function (previousState = {}, action) {  
  // 根据 action 和 state.data3 计算产生新的 state.data3  
  return state.data3;  
};
```

最后，利用 combineReducer 将这三个子 reducer 函数合并并返回完整归一的 finalReducer。

```
const { combineReducers } = Redux;  
const finalReducer = combineReducers({  
  data1: reducer1,  
  data2: reducer2,  
  data3: reducer3  
});
```

在 ES 6 开发环境下,常用的做法是令子 reducer 函数名称与数据状态命名一致,即将 reducer1、reducer2、reducer3 分别命名为 data1、data2 和 data3。

```
const finalReducer = combineReducers({
  data1: data1,
  data2: data2,
  data3: data3
});
```

于是,就可以简写为:

```
const finalReducer = combineReducers({data1, data2, data3});
```

总之,将这三个 reducer 合并成一个 finalReducer。在日常开发中,我们可以单独维护这三个子 reducer,这样的分治会带来维护效率的提升,如图 3-5 所示。

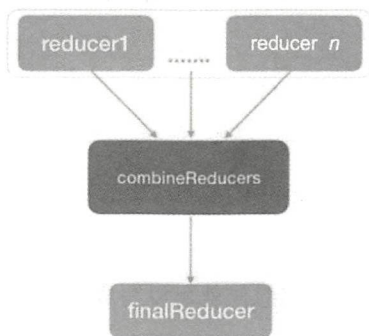


图3-5 合理拆分reducer函数示意图

我们再回过头来看一下 combineReducers 这个方法,它接收一个 Object 类型的参数,这个参数定义了页面数据状态中不同的数据部分与更新这些数据的 reducer 函数之间的映射关系,并最终返回一个合并完整的 reducer 函数。

在实际开发过程中,我们往往遵循着这样一个“最佳实践”:将 reducer 命名为其处理的页面状态数据树中的键值。比如有如下页面状态数据树:

```
const state = {
  data1: {...},
  data2: {...},
  data3: {...}
}
```

同时, `reducer1`、`reducer2`、`reducer3` 分别处理 `data1`、`data2` 和 `data3`, 那么我们同样将 `reducer1`、`reducer2`、`reducer3` 分别命名为 `data1`、`data2` 和 `data3`。

```
const data1 = function (state.data1, action) {  
  ...  
}  
const data2 = function (state.data1, action) {  
  ...  
}  
const data3 = function (state.data1, action) {  
  ...  
}
```

这样做的好处是 `reducer` 命名更加规范和清晰, 能够准确表达其处理对应数据的意义, 同时在多人开发时也有利于维护。在这种情况下, 我们同样可以利用 ES 6 中的 `Object` 新特性对 `combineReducers` 进行简写。

```
const finalReducer = combineReducers({  
  data1,  
  data2,  
  data3  
});
```

还记得创建 `store` 实例的方式吗? 这时候就可以写成:

```
const store = createStore(combineReducers(...), preloadedState, enhancer);
```

或者

```
const store = createStore(finalReducer, preloadedState, enhancer);
```

本节中我们介绍了 `Redux` 的基本使用和实践, 希望读者结合前面所介绍的 `Redux` 思想进行体会、学习。到目前为止, 我们只是分块对 `Redux` 进行了解读, 就像学习一套广播体操一样, 我们介绍了不同运动小节的动作, 最后还需要将不同的运动小节融合、衔接, 这样才算是学会了广播体操。

3.5 Redux 开发基础实例

前面我们介绍了 `Redux` 的基本使用环节, 相信读者已经能够实现简单开发了, 但是可能会感到无从下手, 这是因为还需要打通“任督二脉”, 将所有的知识点进行连接。本节将通过一个非常简单的开发实例, 贯穿 `Redux` 架构流程, 进而进一步加深理解。

Redux 是独立的，可以与很多框架或类库一起使用。本节将用最基本的原生 JavaScript（不包含 React）来演示 Redux 架构。关于 Redux 和 React 的结合，后续章节会进行介绍。

以下是一个非常常见的场景，页面截图来自百度经验产品。页面上有一个点赞组件，包含两个按钮：“赞”和“踩”，以及点赞数，如图 3-6 所示。



图3-6 页面上的点赞组件

点击“赞”按钮，页面上展示点赞数增加 1；点击“踩”按钮，页面上展示点赞数减少 1。

现在来看看在 Redux 架构下应该如何实现。

首先引入依赖：

```
import { createStore } from 'redux';
```

前面介绍过，store 是 Redux 架构容器，它保存着当前页面数据状态，提供 dispatch 方法，派发开发者定义的 action。那么对于这样的开发需求，需要用到哪些 action 呢？其实很简单，直观来讲就是：赞 action 和踩 action。

对于这两种 action，因为已经明确说明“总赞数增加 1 或减少 1”，且页面需求只会进行增减 1 的操作，所以并不需要额外的数据信息。于是，我们分别定义：{ type: 'LIKE' } 为赞 action；{ type: 'UNLIKE' } 为踩 action。

当创建一个 store 时，需要编写 reducer 函数来进行页面数据状态的变更，reducer 负责根据相应的 action 和 previousState 推算出最新的数据状态。同时需要注意：reducer 处理方式必须保持纯函数的纯净特性。

60 React 状态管理与同构实战

```
const reducer = (previousState = 0, action) => {
  switch (action.type) {
    case 'LIKE':
      return previousState + 1;
    case 'UNLIKE':
      return previousState - 1;
    default:
      return previousState;
  }
}
```

这样就具备了创建 store 的条件，使用 createStore 方法并传入 reducer。

```
const store = createStore(reducer);
```

为了简单起见，这里并未设置 createStore 方法的第二个参数 preloadedState 来描述数据初始状态。

接下来，我们进行测试，获取当前状态。

```
console.log(store.getState()); // 0
```

执行以上语句得到当前点赞数为 0，因为我们在 reducer 中设置了初始值 0。

再来测试 dispatch 方法。

```
store.dispatch({ type: 'LIKE' });
```

派发了一个 'LIKE' action，Redux 会自动将这个 action 交给我们提前写好的 reducer 函数处理。

此时，再次获取数据状态，得到点赞数为 1。

```
console.log(store.getState()); // 1
```

至此，state 已经能够跟随着 action 进行变更了。

为了使点击“赞”或者“踩”按钮后能够自动触发页面更新，我们需要使用 store.subscribe 方法，该方法接收一个响应函数作为参数，这个响应函数会在每次派发 action 并更新完状态之后被调用。

点击“赞”按钮，派发 action。

```
document.querySelector('.add-btn').addEventListener('click', () => {
  store.dispatch({ type: 'LIKE' });
});
```



上面的代码会使得点击“赞”按钮后得到新的状态。在更新数据后，更新页面。

```
const render = () => {  
  document.querySelector('.like-num').innerText = store.getState();  
};  
  
store.subscribe(render);
```

这个例子并没有涉及 `combineReducers` 等的用法，但是比较清楚地诠释了 Redux 的思想。下面我们再通过图 3-7，进一步理解到目前为止所掌握的 Redux 整体流程。

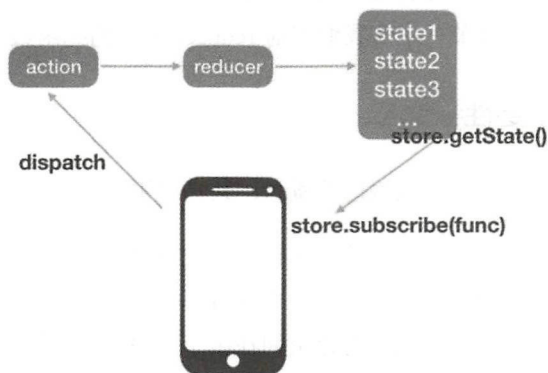


图3-7 Redux整体流程图

其中，`getState`、`subscribe`、`dispatch` 部分是 Redux 提供给开发者使用的方法或 API。另外，`action`、`reducer` 是在 Redux 范式要求下，开发者需要根据业务需求自己独立完成的部分。在整个流程中最核心的 `store` 对象，也是开发者在应用开发中创建的。

Redux 完全可以独立于 React 而存在。它的架构很简单：需要开发者预先定义 `action` 及 `reducer` 函数，同时在恰当的时候派发 `action`，即 `dispatch(action)`。如果所编写的 `reducer` 函数没有问题，那么在正常更新状态之后，就可以通过 `store.subscribe` 注册每一次数据更新后的回调逻辑，这种回调逻辑往往就是对页面的渲染，也就是以后 React 需要衔接的环节。在回调逻辑中，使用 `store.getState()` 获取最新数据，完成正确的页面响应。使用过 React 搭配 Redux 开发的读者可能对 `store.subscribe` 有些陌生，因为它已经由 `react-redux` 库进行了封装，这也是 `store` 数据更新后便可以直接触发相关组件重新渲染的原因。

看到这里，读者可能会觉得：“Redux 貌似一个发布订阅系统”。其实的确如此，但也不完全准确。它的实现确实是一个发布订阅系统，但是其思想在发布订阅模式的基础上又多了函



数式和更多的架构范式。阅读后续章节，参考其源码分析，你会发现更多的奥秘。

3.6 reducer 编写关键：不可变性

通过前面的介绍我们了解到：reducer 函数需要保持纯函数的特性。很多前端开发者对“纯函数”这个函数式编程里的概念可能很陌生，同时也很难说 JavaScript 语言完全适合函数式操作。当然，由于 JavaScript 的灵活性，它还为纯函数的实现提供了一些便利。

另外，使用 React 搭配 Redux 开发应用，非常关键的一步是对数据状态的准确管理，这也是保证程序良好运行的重要环节。本节我们就对 reducer 函数的纯函数操作进行更深入的讲解。

3.6.1 共享和不可变性

我们有必要先来认识两个函数式编程的概念：共享和不可变性。

共享和不可变性是函数式编程推崇的重要概念，也是其显著特点。保证数据的不可变性，好处在于：开发更加简单、可回溯、测试友好，以及减少了任何可能的副作用，从而减少了 Bug 的出现。共享是指一个变量、对象或者内存空间在多个共享的作用域中出现，或者一个对象的属性在多个作用域范围内被传递。共享带来的问题是：针对共享的数据，我们需要完全掌握其所有作用域空间内的情况，以保证代码的正确性。

我们还是从 JavaScript 基础说起。在 JavaScript 中，数据类型包括：

- 基本数据类型，如 undefined、null、number、boolean、string 等。
- 引用类型，如 Object、Array、Function 等。

一般基本数据类型都保存在栈内存当中，引用类型都保存在堆内存当中。所以有如下结论：

- 基本数据类型的值是不可变的。
- 引用类型的值是可变的。

这样讲解有些抽象，我们来看具体的代码示例。

```
var a = 10;
var b = a;
a++;
```



```
console.log(a); // 11  
console.log(b); // 10
```

首先声明第一个基本数据类型变量 `a`，并赋值为 10，如图 3-8 所示。

栈内存空间

变量	值
a	10

图3-8 基本数据类型变量a存储示意图

然后声明第二个基本数据类型变量 `b`，当 `a` 给 `b` 赋值时，会在内存中分配全新的地址，并得到相应的值，如图 3-9 所示。

栈内存空间

变量	值
a	10
b	10

图3-9 基本数据类型变量a、b存储示意图（1）

变量 `a` 的值增加 1 之后，并不会影响 `b` 的值，如图 3-10 所示。

栈内存空间

变量	值
a	11
b	10

图3-10 基本数据类型变量a、b存储示意图（2）

这就是为什么说基本数据类型的值是不可变的。还记得我们在上一节中介绍的 `reducer` 函数吗？

```
const reducer = (previousState = 0, action) => {  
  switch (action.type) {  
    case 'LIKE':  
      return previousState + 1;  
    case 'UNLIKE':  
      return previousState - 1;  
    default:  
      return previousState;  
  }  
}
```




因为 `previousState reducer` 函数就是纯函数,对于基本类型的增减操作并不会影响其他状态。

对于引用类型,我们声明变量 `c` 和 `d`。

```
var c = [1, 2, 3];
var d = { e: 20 };
```

变量 `c` 存在于栈内存中,但是 `[1, 2, 3]` 作为对象其实存在于堆内存中。同样,变量 `d` 存在于栈内存中, `{e: 20}` 作为对象存在于堆内存中。当要访问 `c` 和 `d` 的值时,首先从栈内存中获取 `c` 和 `d` 的引用地址,然后根据这个引用地址再从堆内存中获取所需的值,如图 3-11 所示。

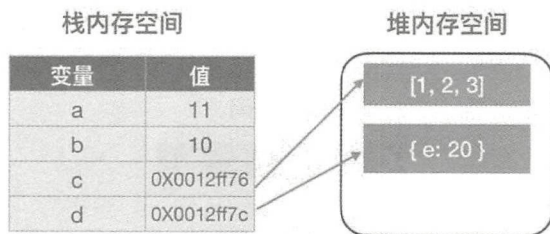


图3-11 引用类型变量c、d存储示意图

下面我们论证引用类型带来的共享问题和可变性。

```
var f = c;
var g = d;
c[0] = 0;
d.e = 30;
console.log(f); // 0, 2, 3
console.log(g); // {e: 30}
```

当引用类型变量 `c` 和 `d` 分别赋值给 `f` 和 `g` 之后, `f` 和 `g` 实际上获得的是 `c` 和 `d` 的引用。这就是一种共享,因此当 `c` 和 `d` 的值发生变化时, `f` 和 `g` 的值也会随之发生变化,如图 3-12 所示。

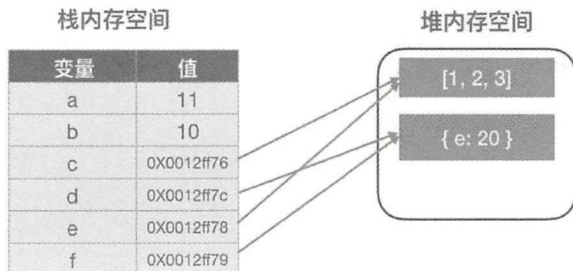


图3-12 复杂数据类型操作后存储示意图



这就是 JavaScript 中的共享和可变性。Redux 秉承函数式，是排斥这种共享和可变性的。这样一来，对于基本数据类型还好处理；但是对于引用类型，应该怎么做以保证不可变性呢？

3.6.2 在 Redux 架构下保证不可变性

通过上面介绍，我们会想到，在 Redux 的 reducer 一次更新过程中，不应该直接更改原有对象或数组的值，因为它们是引用类型，直接更改其值是不被允许的。这时就需要新建一个新的对象或数组用来承载新的数据，以保证纯函数的特性。

对于 JavaScript，因为其灵活性，我们完全可以保证不可变性。下面主要分数组 and 对象两种情况进行解析。

1. 数组操作

如果页面数据状态部分是以数组的形式存储的，对于数据的更新，则主要集中在增、删、改操作上，那么我们如何处理以保证不可变性呢？

(1) 增加一项

使用 push 方法显然不能满足需求，毫无疑问，它改变了原有数组的值。

```
let a = [0,1,2];
a.push(3);
console.log(a); // [0, 1, 2, 3]
```

可以考虑使用 concat 方法，它不会对原有数组进行改动，而是创建一个新的数组。

```
let a = [0,1,2];
let b = a.concat([3]);
console.log(a); // [0, 1, 2]
console.log(b); // [0, 1, 2, 3]
```

我们并未对 a 数组进行修改，而是在 a 数组的基础上生成了一个全新的数组 b。对应 reducer 中，就是：

```
let array = [1, 2, 3];
const addArrayReducer = (array, action) => {
  return array.concat(action.data);
}
let newArray = addArrayReducer(array, {type: 'ADD', data: [4]});
```



```
console.log(array); // [1, 2, 3]
console.log(newArray); // [1, 2, 3, 4]
```

想象 `addArrayReducer` 是一个 `reducer`，那么其参数 `array` 相当于旧的页面数据状态部分，第二个参数为 `action`。

(2) 删除一项

对于删除某一项的操作，`splice` 也不能满足需求，该方法会改变原有数组的值。相应地，我们应该使用 `slice`，并结合 ES Next 新特性。以删除第 2 项为例：

```
let array = [1, 2, 3];
const removeArrayReducer = (array, index) => {
  return [
    ...array.slice(0, index),
    ...array.slice(index + 1),
  ]
};

let newArray = removeArrayReducer(array, 1);
console.log(array); // [1, 2, 3]
console.log(newArray); // [1, 3]
```

这其实就是根据需要删除的项，对其前后进行切片，最后再拼接成新的数组。

在实际的 `reducer` 编写中，如果 `removeArrayReducer` 是一个 `reducer`，那么其参数 `array` 相当于旧的页面数据状态部分，第二个参数 `index` 往往出现在 `action` 的负载数据中，如 `action.payload`。

(3) 更新一项

在 `array` 数组中，需要将第 2 项增加 1，直接进行 `array[1]++` 操作必定不能满足需求，毫无疑问，它改变了原有数组的值。相应地，思路同前：

```
let array = [1, 2, 3];
const incrementCounter = (array, index) => {
  return [
    ...array.slice(0, index),
    array[index] + 1,
    ...array.slice(index + 1)
  ];
};
```



```
};  
let newArray = incrementCounter(array, 1);  
console.log(array); // [1, 2, 3]  
console.log(newArray); // [1, 3, 3]
```

同样，在实际的 reducer 编写中，如果 incrementCounter 是一个 reducer，那么其参数 array 相当于旧的页面数据状态部分，第二个参数 index 往往出现在 action 的负载数据中，如 action.payload。

综上所述，对于数组保证不可变性，我们使用了 ES Next 新特性，同时合理地选择了 JavaScript 数组方法。这就需要开发者了解哪些方法会改变原有数组的值，哪些方法会生成新的数组而保证不可变性。

2. 对象操作

如果页面数据状态部分存储在一个 JavaScript 对象中，该如何处理呢？想象对某一项的存储结构是这样的：

```
let item = {  
  id: 0,  
  book: 'Learn Redux1',  
  available: false  
}
```

(1) 更新一项

如果把 item.book 这本书归还了图书馆，那么需要将 item.available 设为 true。直接修改明显会破坏不可变性，从而违背纯函数原则。这里我们将使用 ES Next 新特性带来的 Object.assign 方法。

```
let item = {  
  id: 0,  
  book: 'Learn Redux1',  
  available: false  
}  
  
const setItemAvailable = function (sourceItem) {  
  return Object.assign({}, sourceItem, {  
    available: true  
  });  
}
```



```
let newItem = setItemAvailable(item);
console.log(newItem); // {id: 0, book: "Learn Redux1", available: true}
console.log(item); // {id: 0, book: "Learn Redux1", available: false}
```

或者使用对象扩展运算符。

```
let item = {
  id: 0,
  book: 'Learn Redux1',
  available: false
}

const setItemAlreadyLearned = function (sourceItem) {
  return {
    ...sourceItem,
    available: true
  };
}
```

通过这两种方式都可以实现需求。但不管是数组还是对象的这类操作，都属于浅拷贝，如果嵌套层次超出一层，则需要额外处理，下文会对浅处理问题进行分析。

(2) 增加一项

比如给上面的 item 加入“读后评分”，同样可以使用类似的方式进行处理。

```
let item = {
  id: 0,
  book: 'Learn Redux1',
  available: false
}

const addItemNote = function (sourceItem) {
  return {
    ...sourceItem,
    note: 13
  };
}

let newItem = addItemNote(item);
```



(3) 删除一项

如果需求变动, 不再需要“读后评分”这一项, 那么删除 `item.note` 该怎么做呢?

```
let item = {
  id: 0,
  book: 'Learn Redux1',
  available: true,
  note: 13
}

let newItem = Object.keys(item).reduce((obj, key) => {
  if (key !== 'note') {
    return { ...obj, [key]: item[key] }
  }
  return obj
}, {})

console.log(newItem);
// {id: 0, book: "Learn Redux1", available: true}
console.log(item);
// {id: 0, book: "Learn Redux1", available: true, note: 13}
```

这里使用了 `Object.keys` 及 `reduce` 方法, 对除 `note` 属性以外的所有属性进行累加拷贝。这是很典型的函数式操作。

一个很好的选择是使用类似于 `underscore` 的类库, 这种类库都会对数据操作进行函数式封装。另外, 将原有数据完全深拷贝一份, 然后再对副本进行操作也是一个可选的方案。

(4) 一个关于“深入浅出”的问题

需要注意的是, 使用 `Object.assign` 及扩展运算符等都属于浅操作。如果在 `item1` 外再套一层, 并对 `data.item1.available` 进行更新:

```
let data = {
  item1: {
    id: 0,
    book: 'Learn Redux1',
    available: false
```



```
    }  
  }  
  
  let newData = Object.assign({}, data);  
  newData.item1.available = true;  
  
  console.log(data.item1.available); // true  
  console.log(newData.item1.available); // true
```

这时候问题会渐渐浮现。使用 `Object.assign` 实际上只是做了一层“值拷贝”，对于 `data.item1.available` 拷贝的是其引用。所以上述结果表明，原始数据也被更改了，这就破坏了不可变性。为此，我们需要手动分开所有层分别进行拷贝，实现一种深拷贝。

```
let item1 = Object.assign({}, data.item1);  
let newData = Object.assign({}, {item1});  
newData.item1.available = true;  
  
console.log(data.item1.available); // false  
console.log(newData.item1.available); // true
```

我们也可以借助 `reduce` 方法。

```
let data = {  
  item1: {  
    id: 0,  
    book: 'Learn Redux1',  
    available: false  
  }  
}  
  
let newData = Object.assign({}, data, {  
  item1: Object.keys(data.item1).reduce((result, key) => {  
    if (key === 'available') {  
      result[key] = true;  
    }  
    else {  
      result[key] = data.item1[key];  
    }  
  })  
});  
return newData;
```




```
    }, {})  
  });  
  
  console.log(data.item1.available); // false  
  console.log(newData.newDataItem.available); // true
```

当然，这种解决方案一定不是唯一的，我们只需要遵循不可变性这个大原则即可。以上操作并不优雅，我们封装一个常用的深拷贝工具函数。

```
const type = obj => {  
  var toString = Object.prototype.toString;  
  var map = {  
    '[object Array]': 'array',  
    '[object Object]': 'object'  
  };  
  return map[toString.call(obj)];  
}
```

```
const deepClone = data => {  
  // 先使用 type 函数进行数据类型判断  
  var t = type(data);  
  var o;  
  var i;  
  var length;  
  
  if (t === 'array') {  
    // 数组类型，新建数组  
    o = [];  
  }  
  else if (t === 'object') {  
    // 对象类型，新建对象  
    o = {};  
  }  
  else {  
    // 基本数据类型的值是不可变的，直接返回  
    return data;  
  }  
}
```



```
if (t === 'array') {
  for (i = 0, length = data.length; i < length; i++) {
    o.push(deepClone(data[i]));
  }
  return o;
}
else if (t === 'object') {
  for( i in data) {
    o[i] = deepClone(data[i]);
  }
  return o;
}
}
```

`type` 函数对拷贝对象进行判断，分为数组和对象两种情况。按照递归思路，进行深拷贝实现。当然，这只是一个示例，并没有对 `object HTMLDivElement` 等其他复杂类型进行处理。另外，实现深拷贝也有其他方式，比如使用 `JSON.stringify` 配合 `JSON.parse`。

```
var newObject = JSON.parse(JSON.stringify(oldObject));
```

示例代码如下：

```
let data = {
  a: "a",
  b: "b",
  c: [
    {
      c11: "c11",
      c12: "c12"
    },
    {
      c21: "c21",
      c22: "c22"
    },
  ],
};

var data1 = JSON.parse(JSON.stringify(data));
data1.c[0].c11 = "c11-changed";
console.log(data.c[0].c11); // c11
console.log(data1.c[0].c11); // c11-changed
```

扩展问题：在实际开发中，如果数据有多层该如何处理呢？深拷贝过程对于开发性能并不友好。另外，在 JavaScript 中满足不可变性操作的方法也有限，有时候难免捉襟见肘。为了解决此类问题，社区中活跃着很多不可变的数据操作类库，如 Facebook 的 `immutable.js`、`mori.js` 等。使用它们实现的深拷贝及数据结构，在注重性能的同时也提供了方便实用的 API。但是引入这些第三方类库也增加了一定的复杂度和学习成本。尤其是在同一个应用中，原生 JavaScript 数组、对象和这些类库带来的新的数据结构混杂，往往也存在着一定程度的混乱。

如果开发者觉得使用 JavaScript 的 `slice`、`filter`、`map`、`reduce` 等函数式 API，再结合 ES Next 新特性已经完全可以满足开发需求，那么就没必要再使用类似于 `immutable.js` 的类库了。在取舍之间，开发者需要根据自身的业务情况及团队风格进行选择。

3.7 Redux 中间件和异步

到目前为止，我们已经掌握了 Redux 的基本用法，并通过简单的例子进行了演示。但是在实际开发中，需求往往复杂多样，为此 Redux 提供了一套中间件机制，使开发者可以在派发任何一个 action 和执行 reducer 这两步之间，添加自定义扩展功能。

对于有服务端编程经验的开发者来说，对中间件（Middleware）这个名词并不陌生。顾名思义，中间件作为中间设备、中间桥梁，它可以连接两种事务或服务。比如对于服务端来说，中间件经常被嵌入在从框架接收请求到产生响应过程之中。因为在这个环节，我们可以利用中间件完成扩展以适应不同的业务需求。

就如开篇所讲的，Redux 中间件提供的是位于 action 被派发之后，到达 reducer 之前的扩展点，因此我们可以利用 Redux 中间件来完成日志记录、调用异步接口或者路由等。

如图 3-13 所示，中间件可以在 action 到达 reducer 之前进行日志记录、中断 action 触发，甚至修改 action，或者不进行任何处理。在 Redux 架构中可以接入多个中间件，这些中间件扮演的增强功能角色由第三方社区开发，当然也可以由开发者自己编写。

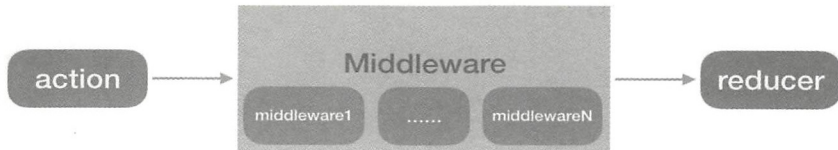


图3-13 中间件角色图

下面我们来认识一些常用的中间件，并加以应用。

Redux 本身提供了 `applyMiddleware` 方法用来接入中间件。创建 `store` 实例的代码如下：

```
const store = createStore(reducer, preloadedState, enhancer);
```

我们可以在 `enhancer` 参数的位置接入中间件。`createStore` 方法可以接收 `applyMiddleware(...middlewares)` 作为参数，创建一个应用了中间件之后的 `store`。

```
const store = createStore(  
  reducer,  
  preloadedState,  
  applyMiddleware(middleware)  
);
```

或者

```
const store = createStore(  
  reducer,  
  applyMiddleware(middleware)  
);
```

关于中间件的具体用法，我们还是通过实例来理解吧。

3.7.1 应用 `redux-logger` 中间件

社区中的 `redux-logger` 中间件用来进行日志记录。每一次派发前后的页面数据状态，以及当前所派发的 `action`，都能在浏览器开发面板中打印出来，方便我们开发调试。

首先，引入 `redux-logger` 中间件，需要开发者先安装依赖。

```
import createLogger from 'redux-logger';
```

然后，利用 `applyMiddleware` 和 `createStore` 接入中间件。

```
import { applyMiddleware, createStore } from 'redux';  
const logger = createLogger();  
  
const store = createStore(  
  reducer,  
  applyMiddleware(logger)  
);
```

这里在 `createStore` 方法中省略了 `preloadedState` 参数。如果需要在创建 store 时进行 store 状态的初始化，那么需要调整 `createStore` 函数，`applyMiddleware(...arguments)` 将会作为第三个参数出现。

```
const store = createStore(
  reducer,
  preloadedState,
  applyMiddleware(logger)
);
```

做完了这些，便接入了 `redux-logger` 中间件，其使用非常简单。在 `redux-logger` 中间件的官方网站上，我们可以看到每一个 action 前后的记录均会出现在控制台中，如图 3-14 所示。

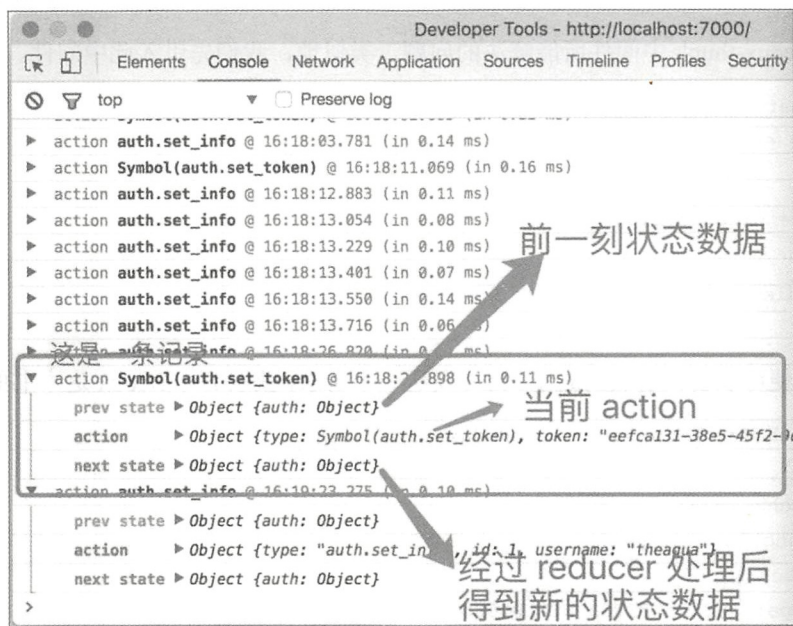


图3-14 redux-logger中间件效果图

3.7.2 应用 `redux-thunk` 中间件

不知道读者是否想过：在 Redux 架构下如何处理异步场景呢？假如有一个异步需求，比如需要派发一个网络请求 action，在网络请求返回之后，再派发一个 action 用来根据返回的数据渲染页面。

对应于 Redux 思想，就是派发一个异步 action。但这是很难做到的，因为 dispatch 的默认参数只能是一个 JavaScript 对象。

如果先派发一个发送请求的 action，再派发一个处理请求返回结果的 action：

```
dispatch(sendRequestAction);  
dispatch(handleResponseAction);
```

这样明显不能满足需求。因为它们是同步进行的，在第一个 dispatch 发送请求还没返回之前，会直接运行第二个 dispatch，此时显然拿不到所需的数据。

那么该怎么做呢？问题出在 dispatch 的参数上。设想一下：如果 dispatch 可以接收一个函数作为参数，在函数体内进行异步操作，并在异步完成后再派发相应的 action，那么便能解决问题。

这就是 redux-thunk 中间件所能解决的问题。类似地，我们先引入此中间件。

```
import { createStore, applyMiddleware } from 'redux';  
import thunk from 'redux-thunk';
```

```
const store = createStore(  
  reducer,  
  applyMiddleware(thunk)  
);
```

接下来，便可以尝试解决问题了。比如发送一个请求，获取 learnRedux 这本书的相关信息。

```
store.dispatch(fetchNewBook('learnRedux'));
```

```
function fetchNewBook (book) {  
  return function (dispatch) {  
    dispatch({  
      type: 'START_FETCH_NEW_BOOK',  
      data: book  
    })  
    ajax({  
      url: `/some/API/${book}.json`,  
      type: 'POST',  
      data: {  
      }  
    }).then(function (bookData) {  
      dispatch({  
        type: 'FETCH_NEW_BOOK_SUCCESS',  

```



```
      data: bookData
    })
  });
}
}
```

显然这里已经能够给 `dispatch` 函数传一个异步函数 `fetchNewBook` 了，这就是 `redux-thunk` 中间件对 `dispatch` 功能的一个增强。

现在，我们来梳理一遍整个过程。首先发送一个异步 action，代码如下：

```
store.dispatch(fetchNewBook('learnRedux'));
```

`fetchNewBook` 函数的参数即为目标书名。

再看看 `fetchNewBook` 函数做了什么。先派发一个 `type` 值为 `START_FETCH_NEW_BOOK` 的 action 来表示请求即将发出。

```
dispatch({
  type: 'START_FETCH_NEW_BOOK',
  data: book
})
```

此时，页面可以显示加载提示等，进行相关渲染。当捕获 `START_FETCH_NEW_BOOK` 这个 action 时，进行页面状态更新，这是在相关的 `reducer` 函数中完成的。

接下来，我们使用 `Promise` 风格的 `AJAX` 来发送异步请求，并在成功获取结果后，派发一个 `type` 值为 `FETCH_NEW_BOOK_SUCCESS` 的 action。同样，对于此 action 的响应也是在相关的 `reducer` 中完成的，`reducer` 根据 `action.data` 即返回结果 `bookData` 进行页面状态更新。

整个过程如图 3-15 所示。

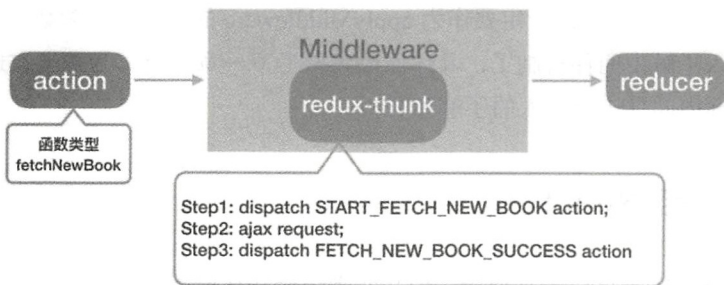


图3-15 redux-thunk中间件角色图

由此不难发现，`redux-thunk` 对于异步处理的关键在于：使 `dispatch` 能够接收异步函数，之后一切变得熟悉而灵活起来，我们完全可以控制 `dispatch` 响应 `action` 的时机。

3.7.3 组合使用中间件

社区中有很多优秀的中间件可供使用，以完成不同功能的增强。`Redux` 中间件最优秀的特性就是可以被组合使用，因此，我们可以在一个项目中使用多个独立的第三方中间件。

比如组合使用 `redux-logger` 和 `redux-thunk` 中间件的情况如下：

```
import { createStore, applyMiddleware } from 'redux';
import thunk from 'redux-thunk';
import createLogger from 'redux-logger';
const logger = createLogger();

const store = createStore(
  reducer,
  applyMiddleware(thunk, logger)
);
```

在上面代码中，`applyMiddleware` 方法的两个参数就是两个中间件。

需要注意的是，有的中间件有次序要求，即 `applyMiddleware` 的参数顺序有讲究。比如，在 `redux-logger` 中就有明确说明：

注意：`logger` 中间件必须位于所有中间件链的最后，否则将无法打印出准确的 `action` (Note: `logger must be the last middleware in chain, otherwise it will log thunk and promise, not actual actions.`)。

所以在上面代码中，`logger` 一定要作为 `applyMiddleware` 的最后一个参数。这是为什么呢？这就需要我们深入了解中间件机制了，具体原因请参考下一章内容。开发者在选用第三方中间件时，应该对目标中间件进行深入的分析和了解。

3.8 Redux 与 React

`Redux` 和 `React` 完全可以独立使用，但是两者结合往往能够发挥最大效力。本节我们就来介绍将这两者结合使用的情况。

3.8.1 Redux 架构回顾

在进入主题之前，我们先对 Redux 做一个简单的总结。一般在使用 Redux 时，我们将整体思路细化为以下几个部分。

- 确定所需的 `state` 数据。
- 根据交互和业务需求，分析确定 `action`。
- 根据不同的 `action`，完成 `reducer` 函数的编写。
- 根据 `reducer` 等，创建 `store`。
- 订阅数据更新，完成视图渲染。

在进行更深入的介绍之前，我们再进行一次梳理。在 Redux 应用中，需要创建一个 `store`，用来存放应用中必要的 `state`。在应用中应该有且仅有一个 `store`。创建 `store` 的方式如下：

```
let store = createStore(reducer, [initialState], enhancer);
```

`store` 是一个 JavaScript 对象，它含有以下几个方法。

- `store.getState()`
- `store.dispatch(action)`
- `store.subscribe(listener)`
- `store.replaceReducer(nextReducer)`

其中，`store.getState()` 用来返回应用当前的页面数据状态，它在 `store` 中进行维护。`store.dispatch(action)` 用来派发 `action`，这是引起页面数据状态更新的唯一途径。它引发 `reducer` 函数，根据 `action` 处理当前的状态。`store.dispatch` 的返回值为相关 `action`，而 `action` 是描述应用变化的 JavaScript 对象，必须含有有效的 `type` 属性。

使用 `createStore` 创建的 `store` 只支持普通对象类型的 `action`。但是如果使用 `applyMiddleware` 作为 `createStore` 的参数，那么中间件可以修改 `action` 的下一步执行操作。中间件是由社区创建的，并不存在于 Redux 源码中，我们需要手动安装。

3.8.2 Redux 和 React 衔接点

下面进入 Redux 和 React 结合的环节。

熟悉 React 的读者，应该知道将 React 应用挂载到页面上的做法为：

```
ReactDOM.render(<RootComponent />, document.getElementById('root'));
```

其中，RootComponent 是根组件，挂载在 id 为 root 的节点上。在使用 Redux 的情况下，页面状态数据全部存储在 store 当中，并通过 store.getState() 获取。

```
const { createStore } = Redux;
const store = createStore(reducer);
```

这时候，RootComponent 组件需要获取页面状态数据，并向下进行派发。这样，store.getState() 的返回值就需要传递给 RootComponent 组件，作为其 props 存在。同时，有了数据之后，根组件也需要感知 dispatch 方法，以处理相关组件引发的交互。相关的事件处理逻辑也需要作为 props 使用 store.dispatch(action) 传递给 RootComponent。

```
ReactDOM.render(
  <RootComponent
    value = {store.getState()}
    onAction = {() =>
      store.dispatch({
        type: 'action1'
      })
    }
  />,
  document.getElementById('root')
);
```

每次 store 有数据更新时，我们都要对相关组件得到的正确数据进行渲染，这就需要再次调用 ReactDOM.render 方法。为了更优雅地实现，我们可以新建一个 render 方法，将 ReactDOM.render 包含其中。每次 store 发生变化时，订阅并调用 render 函数。

```
const render = () =>{
  ReactDOM.render(
    <RootComponent
      value = {store.getState()}
      onAction = {() =>
        store.dispatch({
```

```
        type: 'action1'
      })
    }
  />,
  document.getElementById('root')
);
}
```

```
store.subscribe(render);
render(); // 用于第一次挂载执行
```

到目前为止，我们通过 `store.subscribe` 方法订阅了 `store` 的变化，并且响应此变化，执行 `ReactDOM.render`，同时更新后的 `store` 状态作为根组件的 `props`，经过 `RootComponent` 根组件向下派发，触发相关组件进行重新渲染。

React 和 Redux 衔接示意图如图 3-16 所示。

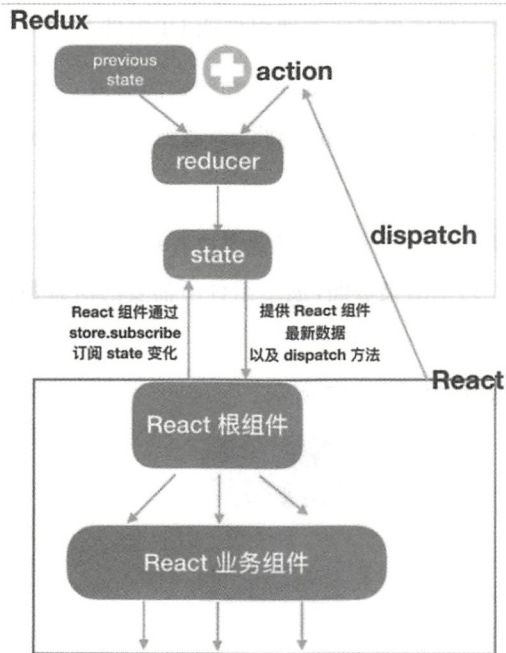


图3-16 React和Redux衔接示意图

Redux 和 React 通过典型的发布订阅模式进行连接。

3.8.3 使用 react-redux 库

上面介绍的原始朴素的做法带来了一些不便。比如增加了很多样板代码，同时对于 React 和 Redux 连接处数据的传递也不够简捷、优雅。更重要的是，store 和组件耦合在一起，包括对状态的获取、触发 action 需要的 dispatch 及 store 的 subscribe 方法等都没有一个更加清晰的包装。

更为普遍的做法是使用 react-redux 库，避免在代码中直接使用 store.subscribe、store.getState。顾名思义，react-redux 对 React 和 Redux 进行了连接，事实上是对上述方法进行了封装和增强，使开发者使用起来更加便捷、高效。目前这个库已经成为 React 和 Redux 开发的必备项目之一。

在学习 react-redux 之前，首先要明确两个重要概念。

- 容器组件（Container Component）
- 展示组件（Presentational Component）

在社区中有的地方将展示组件也称作木偶组件或 UI 组件。

容器组件：所谓容器，实际上是指数据状态和逻辑的容器。它并不负责展示，而是只维护内部状态，进行数据分发和处理派发 action。因此，容器组件对 Redux 是感知的，可以使用 Redux 的 API，比如 dispatch 等。

展示组件：与容器组件相反，展示组件只负责接收相应的数据，完成页面展示，它本身并不维护数据和状态。实际上，为了渲染页面，展示组件所需要的所有数据都由容器组件通过 props 层层传递下来。

展示组件和容器组件示意图如图 3-17 所示。

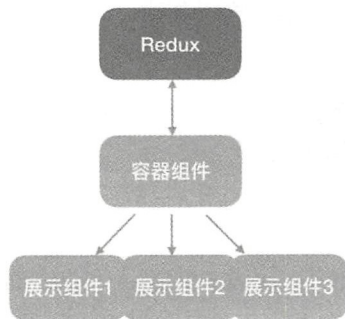


图3-17 展示组件和容器组件示意图

关于这两种组件的对比，Redux 官网上有一个很直观的表格，具体如图 3-18 所示。

	展示组件	容器组件
目的	展示页面内容	处理数据和逻辑
是否感知 Redux	不感知	感知
数据来源	从 props 获取	从 Redux state 订阅获取
改变数据	通过回调 props 派发 action	直接派发 action
由谁编写	开发者	由 react-redux 库生产

图3-18 展示组件和容器组件的对比

那么，通过 react-redux 怎么生成容器组件呢？我们从 react-redux API 看起。首先从最重要的 connect 方法看起。

```
Connect([mapStateToProps], [mapDispatchToProps], [mergeProps], [options])
```

顾名思义，connect 是一个桥梁，用来连接容器组件和展示组件。因为它是一个柯里化函数，对于初学者来说并不好理解。我们要建立起一个概念：connect 的核心是将开发者定义的组件，包装转换生成容器组件。所生成的容器组件能使用 Redux store 中的哪些数据，全由 connect 的参数来确定。

我们来看一下 connect 的几种常见用法，并了解其各种参数的含义。

```
Connect(mapStateToProps, mapDispatchToProps)(presentationalComponent);
```

connect 函数的返回值是一个 WrappedComponent 组件。connect 是典型的柯里化函数，它执行两次，第一次是设置参数；第二次是接收一个正常的 presentationalComponent 组件，并在该组件的基础上返回一个容器组件 WrappedComponent。这其实是一种高阶组件的用法。

我们来认识一下 connect 第一次执行时的两个参数。第一个参数 mapStateToProps 是一个函数，它由开发者设定，其作用是给返回的组件 WrappedComponent 注入 props，这个 props 来自 Redux store 中的状态。所以，这个函数一定要返回一个纯 JavaScript 对象。开发者需要按照下面的形式进行编写：

```
function mapStateToProps ((state, [ownProps])) {  
  return {  
  }  
}
```

它完成从 store 中选取数据并通过 props 传递给将要创建的容器组件 WrappedComponent 的工作。

第二个参数 mapDispatchToProps 的作用是将 dispatch 作为 props 传递给 WrappedComponent 组件。它可以是一个函数，也可以是一个对象。

如果传递的是一个对象，那么键值应该是一个函数，用来描述 action 的生成。也就是说，每个定义在该对象中的函数都将被当作 Redux action creator（构造器），其中所定义的方法名将作为属性名被合并到组件的 props 中。

```
const mapDispatchToProps = { onFirstAct: (data) => {
  type: 'FIRST_ACTION',
  data: data
}};
```

如果 mapDispatchToProps 传递的是一个函数，那么这个函数将接收 dispatch 方法以及容器组件的 props 作为参数，最终也返回一个对象。

```
const mapDispatchToProps = (dispatch, ownProps) => return {
  onFirstAct: () => dispatch({
    type: 'FIRST_ACTION',
    data: ownProps.data
  });
};
```

mapStateToProps 和 mapDispatchToProps 定义了展示组件需要用到的 store 内容。其中 mapStateToProps 负责输入逻辑，就是将状态数据映射到展示组件的参数（props）上；后者负责输出逻辑，即将用户对展示组件的操作映射成 action。

对于它们理解起来相对晦涩，需要开发者在实际应用中去理解，并多加关注。

当不传递 mapDispatchToProps 时：

```
connect(mapStateToProps)(presentationalComponent)
```

返回值为 WrappedComponent。

这里省略了 mapDispatchToProps 参数，在默认情况下，dispatch 方法会注入 WrappedComponent 组件的 props 中。

当两个参数都被忽略时：

```
connect()(presentationalComponent)
```

Redux store 中的状态数据无法传递下来, 因此返回组件 `WrappedComponent` 就不会监听 store 的任何变化。

讲到这里, 也许读者会有疑问: `connect` 是如何获取到 Redux store 中的内容的? 这就要借助于 `Provider` 组件来实现了。`react-redux` 提供了 `Provider` 组件, 一般用法是需要将 `Provider` 作为整个应用的根组件, 并获取 store 为其 prop, 以便后续进行下发处理。

```
<Provider store = {store}>
  <WrappedComponent />
</Provider>
```

这样, `Provider` 组件可以访问 Redux store。

在开发中, 借助于 `react-redux` 的基本模式便是:

```
let WrappedComponent = connect (mapStateToProps, mapDispatchToProps) (presentationalComponent);

ReactDOM.render (
  <Provider store = {store}>
    <WrappedComponent />
  </Provider>
);
```

3.9 实现计数器的四种方式

本节我们将通过四种方式来实现一个简易的计算器。

- 纯 React 实现。
- 纯 Redux 实现。
- React + Redux 实现。
- React + Redux 并使用 `react-redux` 库实现。

通过这个简单的例子, 我们来比较这四种实现方式的不同。

计数器实现示意图如图 3-19 所示。



图3-19 计数器实现示意图

页面计数可以通过四种不同的按钮来改变。

- 计数增加 1。
- 计数减少 1。
- 当当前计数为奇数时，计数增加 1。
- 在点击按钮之后，间隔 1 秒，计数增加 1。

3.9.1 纯 React 实现

这个例子比较简单，我们进行更细致的组件拆分，所有页面内容均由 Counter 组件来呈现。

```
import React from 'react'
import ReactDOM from 'react-dom'
import Counter from './components/Counter'

const rootEl = document.getElementById('root')

ReactDOM.render(
  <Counter/>,
  rootEl
)
```

Counter 组件应该存在一个状态，用于表示当前计数值，以及四个不同的方法，对应四种按钮的动作行为。

```
import React, { Component } from 'react'

class Counter extends React.Component {
  constructor(props) {
    super(props);
    this.state = { value: 0 };
  }
```

```
}

onIncrement = () => {
  this.setState(
    { value: this.state.value + 1 }
  );
}

onDecrement = () => {
  this.setState(
    { value: this.state.value - 1 }
  );
}

incrementIfOdd = () => {
  if (this.state.value % 2 !== 0) {
    this.onIncrement()
  }
}

incrementAsync = () => {
  setTimeout(this.onIncrement, 1000)
}

render() {
  return (
    <p>
      Clicked: {this.state.value} times
      <br/>
      { ' ' }
      <button onClick={this.onIncrement}>
        +
      </button>
      { ' ' }
      <button onClick={this.onDecrement}>
        -
      </button>
    </p>
  );
}
```

```

    { ' ' }
    <button onClick={this.incrementIfOdd}>
      Increment if odd
    </button>
    { ' ' }
    <button onClick={this.incrementAsync}>
      Increment async
    </button>
  </p>
)
}
}

```

这里使用 class 中定义箭头函数的方式对 this 进行了绑定。在组件中，对 this 绑定的方法有多种，各有优劣。

3.9.2 纯 Redux 实现

我们知道，Redux 完全可以脱离 React 而独立存在并使用。事实上，Redux 被独立使用时，就类似于一个“发布订阅系统”。下面来看看用传统的原生 JavaScript 配合 Redux 的实现情况。页面元素如下：

```

<html>
  <head>
    <title>Redux basic example</title>
    <script src="https://unpkg.com/redux@latest/dist/redux.min.js"></script>
  </head>
  <body>
    <div>
      <p>
        Clicked: <span id="value">0</span> times
        <br/>
        <button id="increment">+</button>
        <button id="decrement">-</button>
        <button id="incrementIfOdd">Increment if odd</button>
        <button id="incrementAsync">Increment async</button>
      </p>
    </div>
  </body>
</html>

```



```
</body>
</html>
```

使用 Redux 进行逻辑处理，需要在文档中加入：

```
<script>
function counter(state, action) {
  if (typeof state === 'undefined') {
    return 0
  }

  switch (action.type) {
    case 'INCREMENT':
      return state + 1
    case 'DECREMENT':
      return state - 1
    default:
      return state
  }
}

var store = Redux.createStore(counter)
var valueEl = document.getElementById('value')

function render() {
  valueEl.innerHTML = store.getState().toString()
}

render()
store.subscribe(render)

document.getElementById('increment')
  .addEventListener('click', function () {
    store.dispatch({ type: 'INCREMENT' })
  })

document.getElementById('decrement')
  .addEventListener('click', function () {
```



```
    store.dispatch({ type: 'DECREMENT' })
  })

  document.getElementById('incrementIfOdd')
    .addEventListener('click', function () {
      if (store.getState() % 2 !== 0) {
        store.dispatch({ type: 'INCREMENT' })
      }
    })

  document.getElementById('incrementAsync')
    .addEventListener('click', function () {
      setTimeout(function () {
        store.dispatch({ type: 'INCREMENT' })
      }, 1000)
    })
</script>
```

我们在页面按钮元素的 click 事件中进行了 `store.dispatch` 派发，`counter` 函数相当于 `reducer`，因为 `state` 为基本类型，实现不可变性并不需要更多的处理。

3.9.3 React+Redux 实现

现在，我们使用 `React + Redux` 实现。此时我们将页面中最重要的也是唯一的一个数据状态抽出到 `store` 当中，并引入 `INCREMENT` 和 `DECREMENT` 两种 `action`。`reducer` 函数如下：

```
export default (state = 0, action) => {
  switch (action.type) {
    case 'INCREMENT':
      return state + 1
    case 'DECREMENT':
      return state - 1
    default:
      return state
  }
}
```

下面是创建 `store` 的重要一步。



```
import React from 'react'
import ReactDOM from 'react-dom'
import { createStore } from 'redux'
import Counter from './components/Counter'
import counter from './reducers'

const store = createStore(counter)
const rootEl = document.getElementById('root')

const render = () => ReactDOM.render(
  <Counter
    value={store.getState()}
    onIncrement={() => store.dispatch({ type: 'INCREMENT' })}
    onDecrement={() => store.dispatch({ type: 'DECREMENT' })}
  />,
  rootEl
)
```

订阅 store 的变化。

```
render()
store.subscribe(render)
```

上面的依赖部分提到了 Counter，Counter 组件拥有三个 props。

- value: 用于记录当前计数值，由 store.getState()得到。
- onIncrement: 增加计数方法，直接操作 store.dispatch。
- onDecrement: 减少计数方法，直接操作 store.dispatch。

在 Counter 组件内部，便可以依赖这三个 props 进行渲染。

```
import React, { Component } from 'react'

class Counter extends Component {
  incrementIfOdd = () => {
    if (this.props.value % 2 !== 0) {
      this.props.onIncrement()
    }
  }
}
```



```
incrementAsync = () => {
  setTimeout(this.props.onIncrement, 1000)
}

render() {
  const { value, onIncrement, onDecrement } = this.props
  return (
    <p>
      Clicked: {value} times
      { ' ' }
      <button onClick={onIncrement}>
        +
      </button>
      { ' ' }
      <button onClick={onDecrement}>
        -
      </button>
      { ' ' }
      <button onClick={this.incrementIfOdd}>
        Increment if odd
      </button>
      { ' ' }
      <button onClick={this.incrementAsync}>
        Increment async
      </button>
    </p>
  )
}
```

3.9.4 React+Redux 并使用 react-redux 库实现

上面介绍的三种方法是比较基础的架构方式。当然，我们也可以使用 `react-redux` 库实现。对于此例，虽然有“大炮打苍蝇”的感觉，但是对于理解 `react-redux` 库并熟练使用还是有必要的。

为了区分容器组件和展示组件，我们将页面中的所有按钮抽象成无状态（函数式）组件，该组件将完成两项任务。



- 展示按钮文案，引导用户点击。
- 处理点击逻辑。

其中，展示按钮文案以及点击逻辑将由 `props` 传递，保证按钮职责的单一性。

```
const ButtonAct = (props) => <button onClick={props.onClick}>{props.text}</button>;
```

此时，`Counter` 组件如下：

```
class Counter extends Component {
  constructor(props) {
    super(props);

    this.incrementAsync = this.incrementAsync.bind(this);
    this.incrementIfOdd = this.incrementIfOdd.bind(this);
  }

  incrementIfOdd() {
    if (this.props.value % 2 !== 0) {
      this.props.onIncrement()
    }
  }

  incrementAsync() {
    setTimeout(this.props.onIncrement, 1000)
  }

  render() {
    const { value, onIncrement, onDecrement } = this.props

    return (
      <p>
        Clicked: {value} times
      <br/>
    )
  }
}
```



```
    { ' '}

    <ButtonAct onClick={onIncrement} text="+"/>

    { ' '}

    <ButtonAct onClick={onDecrement} text="-"/>

    { ' '}

    <ButtonAct onClick={this.incrementIfOdd} text="Increment if odd"/>

    { ' '}

    <ButtonAct onClick={this.incrementAsync} text="Increment async"/>

  </p>

)

}

}
```

为了与上一节的例子相区别，此处在此 `constructor` 方法中使用 `bind` 方法对 `this` 进行了绑定。同时，页面的主要信息计数值、增加逻辑、减少逻辑都将通过 `props` 传递下来。

这样一来，`reducer` 函数仍然保持不变。

```
export default (state = 0, action) => {
  switch (action.type) {
    case 'INCREMENT':
      return state + 1
    case 'DECREMENT':
      return state - 1
    default:
      return state
  }
}
```

`store` 依然由如下代码生成。

```
import counter from './reducers'
const store = createStore(counter)
```

在使用 `react-redux` 库的前提下，最重要的是使用 `Provider` 组件以及 `connect` 方法，同时 `connect` 方法中的 `mapDispatchToProps` 和 `mapStateToProps` 也至关重要。首先来看生成逻辑。

```
import { Provider, connect } from 'react-redux';
```




```
ReactDOM.render(  
  <Provider store={store}>  
    <App/>  
  </Provider>,  
  rootEl  
)
```

Provider 组件可以感知 store，store 的全部信息将都作为 Provider 组件的 props 值出现。接下来，最重要的一步就是将 store 中的有效信息派发给 App 组件。App 组件由 connect 生成。

```
import Counter from './components/Counter'  
const App = connect(mapStateToProps, mapDispatchToProps)(Counter);
```

我们传递进去原始的 Counter 组件，并通过 mapStateToProps 和 mapDispatchToProps 对 store 进行筛选，完成将 store 内容传递给 Counter 组件的任务。在使用 react-redux 库的所有项目中，最关键的一步就是对 mapStateToProps 和 mapDispatchToProps 的定义。

那么需要将 store 中的哪些信息传递下去呢？首先是页面的数据状态——计数值，通过 mapStateToProps 来完成。

```
const mapStateToProps = state => {  
  return {  
    value: state  
  }  
}
```

同时，对于页面计数的更改，即派发不同的 action，应该通过 mapDispatchToProps 来完成。

```
const mapDispatchToProps = state => {  
  return {  
    onIncrement: () => {  
      store.dispatch({  
        type: 'INCREMENT'  
      });  
    },  
    onDecrement: () => {  
      store.dispatch({  
        type: 'DECREMENT'  
      });  
    }  
  }  
}
```



把它们合并在一起，代码如下。

```
import React from 'react'
import ReactDOM from 'react-dom'
import { createStore } from 'redux'
import Counter from '../components/Counter'
import counter from '../reducers'
import { Provider, connect } from 'react-redux'

const store = createStore(counter)
const rootEl = document.getElementById('root')

const mapStateToProps = state => {
  return {
    value: state
  }
}

const mapDispatchToProps = state => {
  return {
    onIncrement: () => {
      store.dispatch({
        type: 'INCREMENT'
      });
    },
    onDecrement: () => {
      store.dispatch({
        type: 'DECREMENT'
      });
    }
  }
}

const App = connect(mapStateToProps, mapDispatchToProps)(Counter);

ReactDOM.render(
  <Provider store={store}>
```



```
<App/>  
</Provider>,  
rootEl  
)
```

至此，大功告成。现在，我们结合 react-devtools 来了解一下页面结构：Provider 组件完全感知到了 store 的存在。事实上，store 的所有信息都出现在了 Provider 组件的 props 当中，如图 3-20 所示。

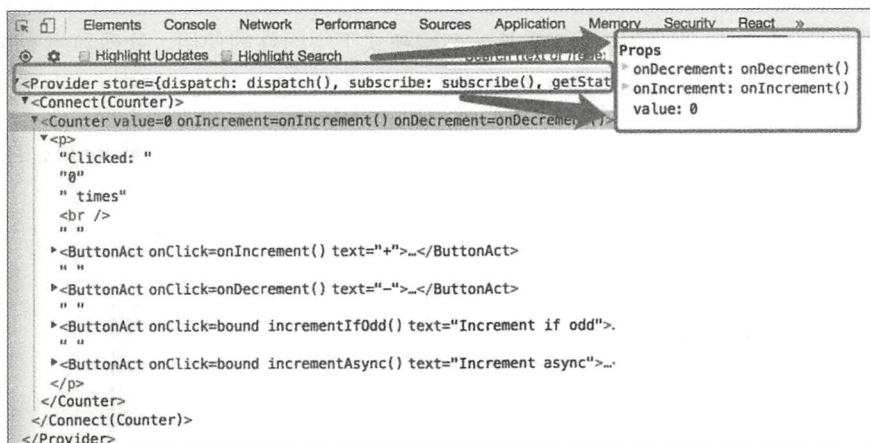


图3-20 Provider组件示意图

通过使用合理的 connect 方法处理过的 Counter 组件，我们得到了所期望的三个 props，如图 3-21 所示。

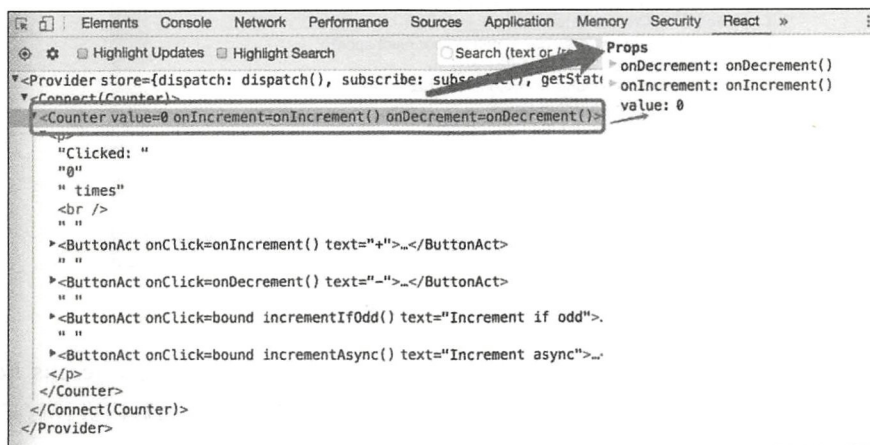


图3-21 Counter组件示意图



其中, value 作为页面计数信息出现; onIncrement 和 onDecrement 都是函数类型, 它们作为 props 经过组件之间的传递, 被页面中按钮的点击动作所引用, 最终触发 dispatch action, 使得数据状态 (value 值) 可以正常更新。

3.10 完成一个工程化实例

本节我们将深入剖析一个工程化实例。该例除涉及 React 和 Redux 的基本使用以外, 还会涉及项目目录设计、组件的合理拆分、Redux 架构模式下中间件的使用、处理异步 action、响应用户交互、前端缓存等内容。这个例子对前面章节中的内容进行了统一梳理。

为了尽量简单, 我们不会对页面进行样式处理, 旨在熟悉 React 搭配 Redux 的架构开发流程。

页面组件设计如图 3-22 所示。

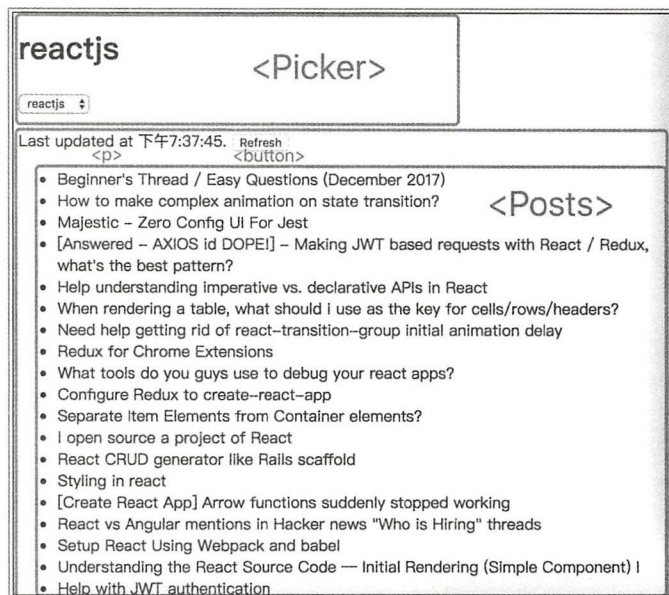


图3-22 页面组件设计图

核心页面展示从服务端获取的数据信息, 用户可以根据头部的菜单选择 reactjs 或 frontend 选项, 如图 3-23 所示。根据用户的选择, 应用将会从服务端获取相应的数据进行展示。

在切换选项时, 如果数据已经存在, 则使用已有数据。在选项菜单下面显示了最新的数据



更新时间。当用户点击“Refresh”按钮之后，将进行数据的更新请求。

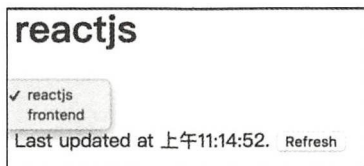


图3-23 选项菜单示意图

在等待请求响应之时，页面将显示一个加载提示，如图 3-24 所示。

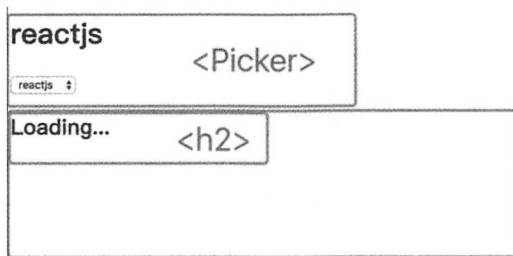


图3-24 Loading设计示意图

这是一个非常朴素的例子，但是囊括了开发流程的诸多方面。接下来我们尝试从分析需求开始，一步一步完成项目的开发。

3.10.1 项目目录

对于一个大型工程实践来说，一个良好、清晰的项目目录至关重要。虽然此项目复杂度并不高，但是良好的目录设计对于后续开发起着非常重要的作用。我们将项目命名为 demo-project，在主目录下，./public 文件夹用于存放 index.html 文件；按照惯例，./node_modules 是所有依赖的目录；所有的 React 和 Redux 业务相关内容存放在 ./src 文件夹中，如图 3-25 所示。

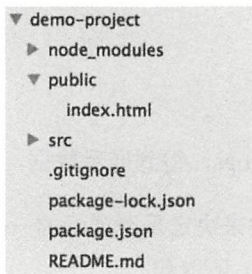


图3-25 项目目录结构图



按照 Redux 的功能区块进行划分，最核心的 src 目录结构如图 3-26 所示。

- ./actions: 维护应用中的所有 action。
- ./components: 维护应用中的展示组件。
- ./containers: 维护应用中的容器组件。
- ./reducers: 定义 reducer 函数。
- index.js: 入口文件。



图3-26 src目录结构图

开发者应根据实际需求以及团队的偏好来设计目录结构。这并没有一个固定模式，原则上只要功能清晰、职责划分合理就行。在大型项目中，如果初期没有合理设定好目录结构，那么在开发过程中就会无比痛苦，比如会出现某个组件或文件位置不合理，添加功能模块混乱、复杂，“牵一发而动全身”等情况。

3.10.2 组件划分

关于组件划分，实际上图 3-26 中已经有了清晰的展示。这里重点说一下容器组件和展示组件的划分。在此项目中，需要展示页面信息的部分由两个展示组件来完成。

- Picker 组件，为选项标签。
- Posts 组件，展示实际内容。

它们通过容器组件传递所需的 props，得到展示数据。组件设计如图 3-27 所示。

事实上，Posts 组件是由判断条件来决定是否展示的——在请求数据尚未得到响应时，Posts 组件的位置将由 Loading 组件（文案）所取代。

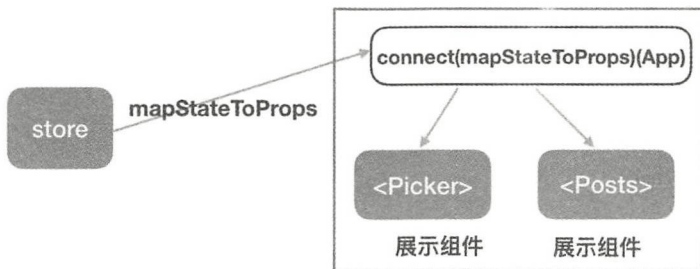


图3-27 组件设计图

3.10.3 确定应用数据状态

应用数据状态是整个架构的核心。当用户初次进入页面时，默认选择展示“reactjs”选项的内容。页面数据结构如图 3-28 所示。

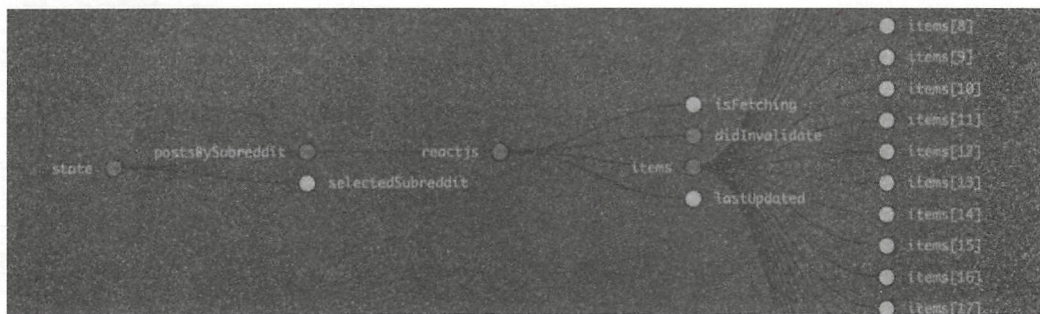


图3-28 页面数据结构图（1）

其中，selectedSubreddit 表示当前选中的选项标签，默认为“reactjs”。另外，该值有可能被用户切换为“frontend”。

postsBySubreddit 存储页面所有的条目信息。当初次进入页面时，只有默认展示的“reactjs”条目。所以 postsBySubreddit 作为对象，在初始时只有“reactjs”条目对应的数据。

- isFetching 表示是否正在请求数据。
- didInvalidate 表达是否更新作废旧数据。
- items 是一个数组，存放相关条目的所有信息。
- lastUpdated 表示最近一次更新时间。



整体的数据结构代码如下：

```
{
  postsBySubreddit: {
    reactjs: {
      isFetching: false,
      didInvalidate: false,
      items: [
        {
          ...
        },
        ...
      ],
      lastUpdated: 1514961189195
    }
  }
}
```

当用户将“reactjs”切换为“frontend”后，页面数据结构如图 3-29 所示。

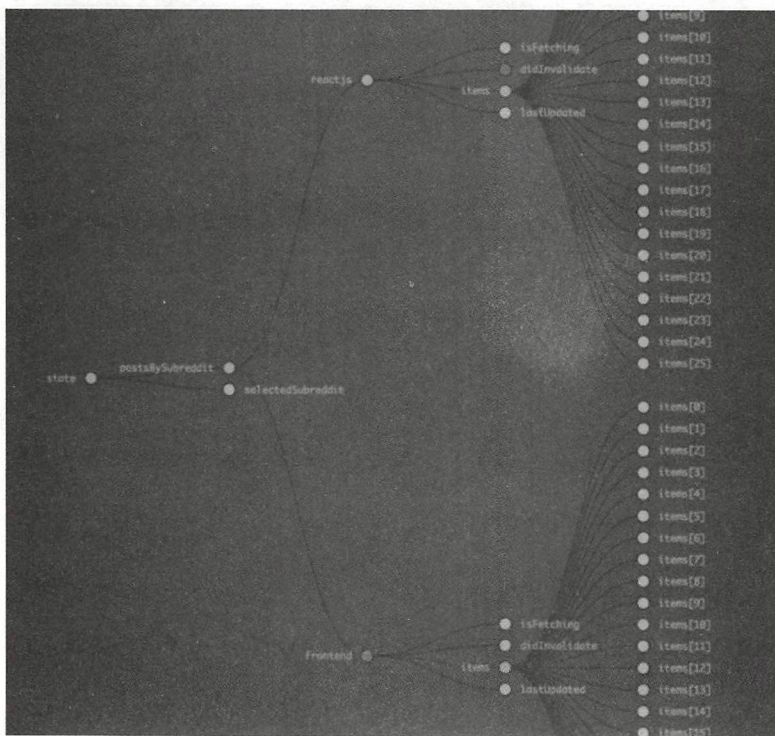


图3-29 页面数据结构图（2）



此时 `postsBySubreddit` 中多了 `frontend` 信息，在这种情况下对应数据快照如下：

```
{
  postsBySubreddit: {
    reactjs: {
      isFetching: false,
      didInvalidate: false,
      items: [
        {
          ...
        },
        ...
      ],
      lastUpdated: 1514961189195
    },
    frontend: {
      isFetching: false,
      didInvalidate: false,
      items: [
        {
          ...
        },
        ...
      ],
      lastUpdated: 1514961189195
    }
  }
}
```

3.10.4 数据状态和相关组件数据分配

明确了页面数据结构后，现在结合 React 组件分析数据应该如何在组件之间流转。Posts 组件展示了相关条目的所有内容，需要得到 `postsBySubreddit[selectedSubreddit]` 信息，即根据选项标签值 (`selectedSubreddit`) 选择相关的 `postsBySubreddit` 内数据。Picker 组件需要获取当前所选择的标签值。



3.10.5 action 和 reducer

接下来,我们思考一下在业务中都会需要哪些 action 值,可分析拆解为 SELECT_SUBREDDIT、REQUEST_POSTS、RECEIVE_POSTS、INVALIDATE_SUBREDDIT。对应代码如下:

```
export const SELECT_SUBREDDIT = 'SELECT_SUBREDDIT';
export const REQUEST_POSTS = 'REQUEST_POSTS';
export const RECEIVE_POSTS = 'RECEIVE_POSTS';
export const INVALIDATE_SUBREDDIT = 'INVALIDATE_SUBREDDIT';
```

其中, SELECT_SUBREDDIT 表示用户在选择框中切换选项的 action,它含有负载信息 subreddit,表示当前的目标选项。该 action 由名为 selectSubreddit 的 action creator 生成,代码如下:

```
export const selectSubreddit = subreddit => ({
  type: SELECT_SUBREDDIT,
  subreddit
})
```

在组件消费层面,它将在用户切换选项时被触发。在 Picker 组件中,设计代码如下:

```
<Picker value={selectedSubreddit}
  onChange={this.handleChange}
  options={[ 'reactjs', 'frontend' ]} />
```

对应的 handleChange 方法如下:

```
handleChange = nextSubreddit => {
  this.props.dispatch(selectSubreddit(nextSubreddit));
}
```

最终由 reducer 处理, reducer 根据 nextSubreddit 值改变数据状态。

```
const selectedSubreddit = (state = 'reactjs', action) => {
  switch (action.type) {
    case SELECT_SUBREDDIT:
      return action.subreddit
    default:
      return state
  }
}
```



也就是说,当用户将选项“reactjs”切换为“frontend”之后,由 reducer 函数将 selectedSubreddit 改为“frontend”。最终 Posts 组件获得了新的 props 数据,利用该数据页面进行相应的渲染更新。

另外,REQUEST_POSTS 和 RECEIVE_POSTS 表示对数据进行请求的过程,其中 REQUEST_POSTS 表示数据请求开始,RECEIVE_POSTS 表示成功获得数据。REQUEST_POSTS 由下面的 action creator 生成。

```
export const requestPosts = subreddit => ({
  type: REQUEST_POSTS,
  subreddit
})
```

具体的请求逻辑在 fetchPosts 函数中:

```
const fetchPosts = subreddit => dispatch => {
  dispatch(requestPosts(subreddit))
  return fetch(`https://www.reddit.com/r/${subreddit}.json`)
    .then(response => response.json())
    .then(json => dispatch(receivePosts(subreddit, json)))
}
```

当然,不是每一次用户交互都会进行数据请求。比如数据已经在请求当中(fetching 值为 true),就需要避免重复发送请求。同时,只有 postsBySubreddit 内不存在相关选项(“reactjs”或者“frontend”)数据时,才需要发送请求,否则使用已有数据。具体由以下逻辑控制:

```
const shouldFetchPosts = (state, subreddit) => {
  const posts = state.postsBySubreddit[subreddit];
  if (!posts) {
    return true;
  }
  if (posts.isFetching) {
    return false;
  }
  return posts.didInvalidate;
}

export const fetchPostsIfNeeded = subreddit => (dispatch, getState) => {
  if (shouldFetchPosts(getState(), subreddit)) {
    return dispatch(fetchPosts(subreddit));
  }
}
```



这里需要注意 `dispatch(fetchPosts(subreddit))` 的参数为 `fetchPosts` 函数，这正是我们使用 `redux-thunk` 中间件来处理异步场景的体现。

同时，我们将 `reducer` 函数拆分为 `postsBySubreddit` 和 `selectedSubreddit`，并使用 `combineReducers` 进行合并。

```
const rootReducer = combineReducers({
  postsBySubreddit,
  selectedSubreddit
})
```

关于 `selectedSubreddit` 前面已经做过介绍。`postsBySubreddit` 是根据请求以及返回的数据进行更新的。

```
const posts = (state = {
  isFetching: false,
  didInvalidate: false,
  items: []
}, action) => {
  switch (action.type) {
    case INVALIDATE_SUBREDDIT:
      return {
        ...state,
        didInvalidate: true
      }
    case REQUEST_POSTS:
      return {
        ...state,
        isFetching: true,
        didInvalidate: false
      }
    case RECEIVE_POSTS:
      return {
        ...state,
        isFetching: false,
        didInvalidate: false,
        items: action.posts,
        lastUpdated: action.receivedAt
      }
  }
}
```




```
    }  
    default:  
      return state  
    }  
  }  
}  
  
const postsBySubreddit = (state = { }, action) => {  
  switch (action.type) {  
    case INVALIDATE_SUBREDDIT:  
    case RECEIVE_POSTS:  
    case REQUEST_POSTS:  
      return {  
        ...state,  
        [action.subreddit]: posts(state[action.subreddit], action)  
      }  
    default:  
      return state  
  }  
}
```

在上面代码中，我们使用了对象扩展运算符来保证数据的不可变性。

3.10.6 使用 react-redux 库进行连接

前文提到，使用 react-redux 库的 connect 方法生成容器组件。

```
export default connect(mapStateToProps)(App);
```

重点是 mapStateToProps 函数的编写。编写原则是根据各个组件所需的不同数据片段进行整合，代码如下：

```
const mapStateToProps = state => {  
  const { selectedSubreddit, postsBySubreddit } = state  
  const {  
    isFetching,  
    lastUpdated,  
    items: posts  
  } = postsBySubreddit[selectedSubreddit] || {  
    isFetching: true,
```



```
    items: []
  }

  return {
    selectedSubreddit,
    posts,
    isFetching,
    lastUpdated
  }
}
```

最终返回 `selectedSubreddit`、`posts`、`isFetching`、`lastUpdated` 四项数据作为 `props` 连通。

这样 `App` 组件便可以在 `render` 方法中调用其子组件，并完成数据传递。

```
render() {
  const { selectedReddit, posts, isFetching, lastUpdated } = this.props;
  const isEmpty = posts.length === 0;
  return (
    <div>
      <Picker value={selectedReddit}
        onChange={this.handleChange}
        options={[ 'reactjs', 'frontend' ]} />
      <p>
        {lastUpdated &&
          <span>
            Last updated at {new Date(lastUpdated).toLocaleTimeString()}.
            { ' ' }
          </span>
        }
        {!isFetching &&
          <a href="#"
            onClick={this.handleRefreshClick}>
            Refresh
          </a>
        }
      </p>
      {isEmpty
```



```

    ? (isFetching ? <h2>Loading...</h2> : <h2>Empty.</h2>)
    : <div style={{ opacity: isFetching ? 0.5 : 1 }}>
      <Posts posts={posts} />
    </div>
  }
</div>
)
}

```

3.10.7 使用中间件和接入 redux-devtools

在此例中，我们使用了两个中间件：redux-thunk 和 redux-logger。创建 store 以及应用的方式如下：

```

import React from 'react'
import { render } from 'react-dom'
import { createStore, applyMiddleware } from 'redux'
import { Provider } from 'react-redux'
import thunk from 'redux-thunk'
import { createLogger } from 'redux-logger'
import reducer from './reducers'
import App from './containers/App'

```

```

const middleware = [ thunk ];
if (process.env.NODE_ENV !== 'production') {
  middleware.push(createLogger())
};

```

```

const store = createStore(
  reducer,
  applyMiddleware(...middleware)
);

```

```

render(
  <Provider store={store}>
    <App />
  </Provider>,
  document.getElementById('root')
);

```

这里要注意中间件的使用顺序。同时，为了使用 `redux-devtools`，我们需要引入 `Redux` 的 `compose` 函数。

```
import { createStore, applyMiddleware, compose } from 'redux';

const composeEnhancers = window.__REDUX_DEVTOOLS_EXTENSION_COMPOSE__ || compose;

const store = createStore(
  reducer,
  composeEnhancers(applyMiddleware(...middleware))
);
```

现在我们通过可视化工具对这个实例进行剖析。首先 `react-redux` 提供的根组件 `Provider` 完全感知到 `Redux store` 的存在，`store` 的所有信息及 API 都存在于其 `props` 中，如图 3-30 所示。



图3-30 Provider组件示意图

`App` 组件通过 `connect` 获得的经过 `mapStateToProps` 处理之后的信息，也作为 `props` 出现在 `App` 组件中，如图 3-31 所示。

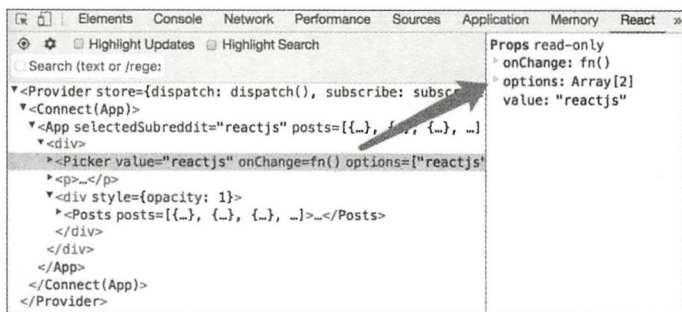


图3-31 Picker组件示意图

`App` 组件同样通过 `props` 将 `onChange`、`options`、`value` 信息传递给 `Picker` 组件，其中 `options` 代表硬编码的“`reactjs`”和“`frontend`”两项，注意它们并不是 `Redux store` 中的相关数据，不需

要 Redux 维护；value 表示当前选中的选项；onChange 是可以触发 dispatch 的回调函数。Picker 组件的信息主要应用在选项标签上，如图 3-32 所示。

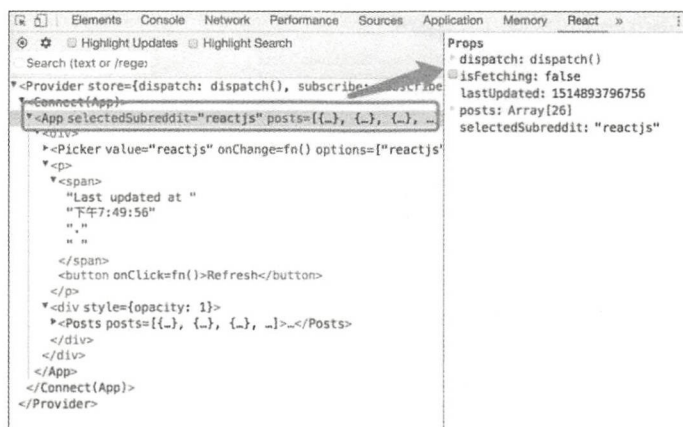


图3-32 App组件示意图

另一个分支，App 组件通过 props 将 posts 数据传递给 Posts 组件，posts 数据由当前选项标签 postsBySubreddit[selectedSubreddit]决定，如图 3-33 所示。

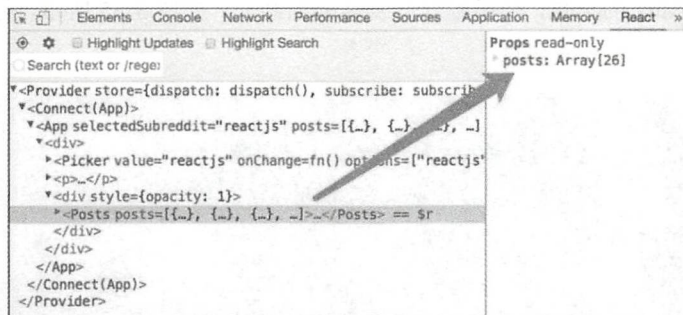


图3-33 Posts组件示意图

Posts 作为函数式组件，也非常简单。

```
const Posts = ({posts}) => (  
  <ul>  
    {posts.map((post, i) =>  
      <li key={i}>{post.title}</li>  
    )}  
  </ul>  
)
```

其中每一项 li 如图 3-34 所示。

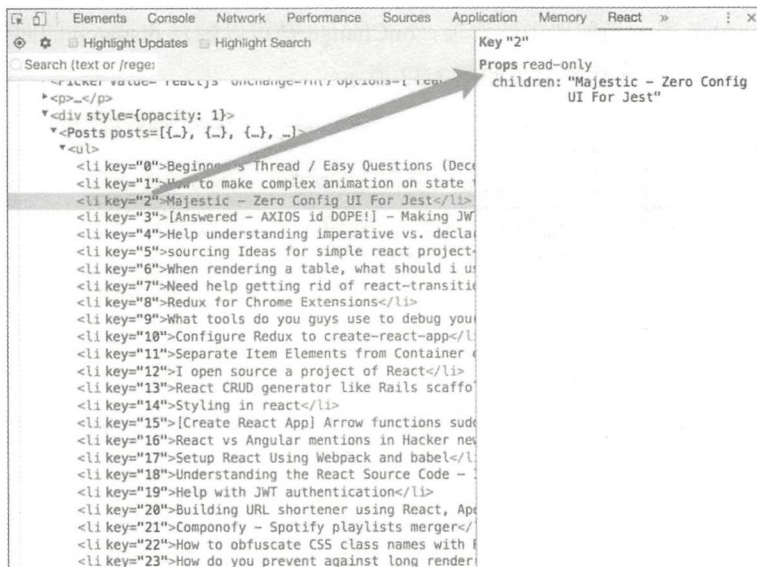


图3-34 props获取示意图

以上截图均来自 react-devtools。同时，强烈建议开发者也接入使用 redux-devtools，这款调试工具可以展现 Redux 的思想，是函数式开发体验的结晶。如图 3-35 所示，每一步的处理都有清晰的记录，十分有利于开发调试。

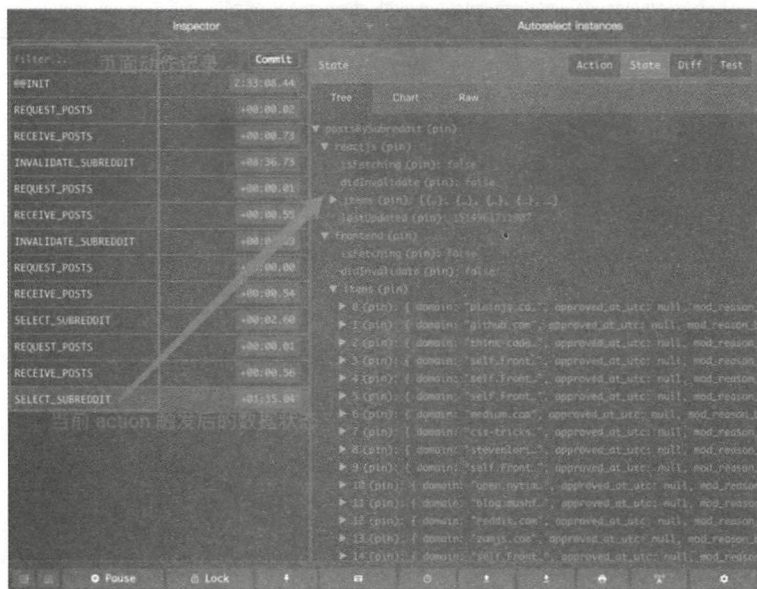


图3-35 Action 触发数据更新示意图

至此，一个简易但“五脏俱全”的项目便分析完成了。

3.11 本章小结

本章着重介绍了 Redux 的设计思想和基本使用方法。了解这些内容后，读者应当建立起对 Redux 最基本的认知，掌握 reducer 函数的编写、派发 action 的设计，以及 store 状态的更新流程。在此基础上，更重要的一个概念是函数式编程。函数式编程具有不可变性、纯净化特性等关键点，这些将会直接决定 Redux 应用开发的质量。为此，本章在数据的不可变性操作上进行了较为深入的实践。同时因为应用开发需求复杂，对于异步处理场景，本章也介绍了 Redux 中间件的使用，Redux 中间件赋予了 Redux 无限可能。

本章还介绍了 React 和 Redux 的结合使用，以及如何使用 react-redux 库进行开发，最后通过实例对前文所讲内容进行了统一梳理，请读者结合本书配套的代码仓库进行体会和学习。

Redux 不是唯一的数据状态管理方案，但是了解和学习它是每一位前端开发者不断进阶的高效方法之一。

第4章

深入理解 Redux

通过前面章节的学习，相信读者已经明白了 React 搭配 Redux 开发应用的大体架构和流程。当然，想要熟练应用还需要实际操练，因为工程化思想积淀一定是建立在丰富的实践经验基础上的。

另外，使用的技术栈越多，在理论层面就会有越多的疑问。比如，Redux 相当于一个发布订阅系统，那么这个系统是如何实现的？Redux 强大的中间件机制基于什么原理？react-redux 库连接 React 和 Redux 的神奇之处在哪里？

虽然这些知识对于基础开发并不是必需的，但是了解这些底层原理，对于提高对 React 和 Redux 架构的整体认知是非常重要的。事实上，开发者想要进阶，想要开发基于 React 或 Redux 的“轮子”，想要构造自己的“中间件”，想要真正加入 React Redux 强大的社区，掌握更深层次的知识是很有必要的。

这一章我们就深入讲解 Redux 原理，了解其源码实现、中间件的设计、react-redux 库的奥秘，以及一些开发最佳实践。

4.1 Redux 源码探索——store 的实现

本节我们来深入分析 Redux 中 store 的源码实现及设计原理。这里不会直接对源码进行讲解，而是循序渐进、从无到有进行模拟和实现。

首先回顾一下创建 store 的方式。在前面章节中，我们使用 Redux 暴露的 createStore 函数来创建 store 实例。

```
const { createStore } = Redux;
```

```
const store = createStore(reducer, preloadedState, enhancer);
```

store 作为一个对象，提供了直接获取页面数据状态的 `getState` 方法、触发更新 store 的 `dispatch` 方法，以及订阅 store 状态变化的 `subscribe` 方法等，进而维护了整个应用的数据状态和页面渲染。如下是 store 的重要方法。

```
store = {
  dispatch,
  getState,
  subscribe,
  replaceReducer
}
```

订阅 store 状态的变化，适时渲染：

```
const render = () => {
  document.body.innerText = store.getState();
};
store.subscribe(render);
render();
```

响应页面交互：

```
document.addEventListener('click', () => {
  store.dispatch({type: 'example'})
});
```

接下来，我们思考如何实现 store。从创建 store 的 `createStore` 函数入手，一般较为简单的情况是它需要接收 `reducer` 函数作为参数（也可以接收其他更多的参数）。

```
const createStore = (reducer) => {
  ...
}
```

我们知道，store 实例持有当前状态。在这里使用一个变量 `state` 来保存应用状态。

```
const createStore = (reducer) => {
  let state;
}
```

同时 `createStore` 返回一个完整的 store 对象，我们需要实现 `getState`、`dispatch` 和 `subscribe` 方法。

```
const createStore = (reducer) => {  
  let state;  
  
  const getState = () => state;  
  
  const dispatch = (action) => {  
    ...  
  }  
  
  const subscribe = (listener) => {  
    ...  
  }  
  
  return { getState, dispatch, subscribe };  
}
```

现在已经有了 `createStore` 函数的基本雏形。接下来，继续完成每一个对外暴露的方法的实现，实际上就是设计模式中的发布订阅模式的简易实现。

我们使用 `listeners` 数组来存储订阅回调函数，这些回调函数用来处理页面组件重新渲染的逻辑。`dispatch` 方法需要触发 `reducer` 函数的执行，进而触发回调函数。

```
const createStore = (reducer) => {  
  let state;  
  
  // listeners 用来存储所有的监听函数  
  let listeners = [];  
  
  const getState = () => state;  
  
  const dispatch = (action) => {  
    state = reducer(state, action);  
    // 每一次状态更新后，都需要调用 listeners 数组中的每一个监听函数  
    listeners.forEach(listener => listener());  
  }  
  
  const subscribe = (listener) => {
```

```
// subscribe 可能会被调用多次，每一次调用时，都将相关的监听函数存入 listeners 数组中
listeners.push(listener);
}

return { getState, dispatch, subscribe };
}
```

这样就实现了订阅函数，那么取消订阅函数该如何实现呢？在 Redux 设计中，subscribe 函数被调用后会返回一个取消订阅函数，在调用此函数时进行取消订阅。类似于：

```
let unsubscribe = store.subscribe(handleChange);
unsubscribe();
```

所以可以像下面这样来实现：

```
const createStore = (reducer) => {
  let state;
  // listeners 用来存储所有的监听函数
  let listeners = [];

  const getState = () => state;

  const dispatch = (action) => {
    state = reducer(state, action);
    // 每一次状态更新后，都需要调用 listeners 数组中的每一个监听函数
    listeners.forEach(listener => listener());
  }

  const subscribe = (listener) => {
    // subscribe 可能会被调用多次，每一次调用时，都将相关的监听函数存入 listeners 数组中
    listeners.push(listener);
    // 返回一个函数，进行取消订阅
    return () => {
      listeners = listeners.filter(item => item !== listener);
    }
  }

  return { getState, dispatch, subscribe };
}
```

在 `subscribe` 方法中返回一个函数用于取消订阅。其实现方式是使用数组的 `filter` 方法，直接过滤掉取消订阅的函数即可。

在实际应用中，`createStore` 被调用后，`Redux` 就会设置一个初始的空状态，我们只需要在 `createStore` 方法体中加入如下一行首次触发一个空的 `action` 即可。

```
dispatch({});
```

至此，我们就实现了 `Redux` 中的 `createStore` 方法，在这个方法中基本包含了除 `replaceReducer` 方法以外的所有特性。如果参照源码，我们会发现源码的实现思路与上述实现方法完全吻合，只不过多了一些对异常进行判断处理的逻辑。如果你明白了上述内容，也一定很容易理解 `Redux` 源码的实现思想。

参考源码，还有一些注意事项或者关键点需要读者注意。

- `@@redux/INIT` 是 `Redux` 保留的私有 `action type`，在业务代码中需要避免使用这个 `action type`。
- 在整个应用中只能存在一个 `store`。
- `action` 类型只支持 `plain object`（`JavaScript` 原生对象），如果开发者想派发 `Promise`、`observable`、`thunk` 函数，或者其他任何形式，则需要使用合适的第三方中间件来完成。
- 本节未实现的 `replaceReducer` 方法很简单，只需要将 `replaceReducer` 方法的参数赋值给 `currentReducer` 即可。需要注意的是，在源码中赋值完成后会再次派发一个类型为“`@@redux/INIT`”的 `action`。
- 为了对一些 `observable/reactive` 库进行兼容，`Redux` 也实现了 `observable` 方法。

4.2 Redux 源码探索——combineReducers 的实现

本节我们按照上一节的思路，实现 `Redux` 中的 `combineReducers` 方法。

本书第 3 章中介绍了 `combineReducers` 方法，它实现了接收多个 `reducer` 函数，并进行整合，归一化成一个 `rootReducer`。

```
const { combineReducers } = Redux;  
const rootReducer = combineReducers({
```



```

    reducer1,
    reducer2,
    reducer3,
    ....
  })

```

其返回值 `rootReducer` 将会成为 `createStore` 的参数，完成 `store` 的创建。

```
const store = createStore(rootReducer, preloadedState, enhancer);
```

使用 `combineReducers` 的好处很明显，开发者可以按照应用状态进行分治，拆分成多个 `reducers`，有利于开发和维护。为了更深入地了解 `combineReducers` 的原理，下面我们从无到有来实现 `combineReducers`。

首先，我们要了解 `combineReducers` 的工作方式。`combineReducers` 只接收一个参数，这个参数阐述了不同 `reducer` 函数和页面状态数据树不同部分的映射匹配关系。

`combineReducers` 的返回值是一个归一化的 `rootReducer` 函数。也就是说，`combineReducers` 作为一个函数，它返回了另一个函数（`rootReducer`）。“函数返回一个新的函数”，这就是函数式编程的典型思想。`combineReducers` 返回的函数就是一个标准的 `reducer`，它的参数接收 `state` 和 `action`，因此有了如下框架。

```

const combineReducers = (reducers) => {
  return (state = {}, action) => {
    ...
  }
}

```

然后，思考 `combineReducers` 返回的函数的返回值应该是什么。`combineReducers` 返回的函数是归一化的 `rootReducer`，其返回值应该是经过各个 `reducers` 计算后的全新页面数据状态，即更新之后的 `state`。

为了获得参数，即所有 `reducers` 的计算结果，我们使用 `Object.keys` 对参数 `reducers` 的 `key` 进行遍历，`Object.keys(reducers)` 返回由所有参数 `reducers` 的 `key` 所组成的一个数组。这样我们就可以根据数组的每一项进行 `state` 的计算，因为这个数组的每一项都是自定义的 `reducer` 函数。考虑到最终的返回结果应该是完整的 `state`，因此使用数组的 `reduce` 方法对 `state` 进行计算并累加。在这种情况下，使用一个空对象作为 `reduce` 方法的计算初始值。

```

const combineReducers = (reducers) => {
  return (state = {}, action) => {

```

```
return Object.keys(reducers).reduce(
  (nextState, key) => {
    nextState[key] = reducers[key](
      state[key],
      action
    );
    return nextState;
  },
  {}
);
}
```

这种实现方式需要读者对数组的 `reduce` 方法等基础内容有比较深入的理解。下面我们再梳理一下。

在调用 `combineReducers` 时，参数是一个对象（在上述实现中名为 `reducers`），`reducers` 这个对象包含很多数据项，每一项的值都是一个 `reducer` 函数，键值是这个 `reducer` 函数所处理的页面状态数据树的名字。

```
{
  statePart1: reducer1,
  statePart2: reducer2,
  statePart3: reducer3,
  ...
}
```

接下来，我们使用了 `Object.keys` 将 `reducers` 这个对象的所有 `key` 抽离为数组，得到：

```
[statePart1, statePart2, ..., statePartN]
```

接着对这个数组使用 `reduce` 方法，在上面代码中 `reducers[key]` 为相关 `reducer` 函数（`reducer1`, `reducer2`...）；`nextState[key]` 则是这个 `reducer` 函数对应的 `state` 部分。最后，在初始值为一个空对象的情况下，对各个 `statePart` 进行计算，得到最终的全局 `state`。

当然，是否理解这些逻辑并不会直接影响对 `Redux` 的使用。但是对于体会函数式编程思想，以及理解“一个函数接收另一个函数作为参数”和“一个函数返回另一个函数”等典型的函数式操作非常重要。

Redux 源码的实现也秉承了上述思想，但是采用了 for 循环来代替 reduce 遍历。请参考本书配套的代码仓库来理解。

4.3 dispatch 的改造——实现记录日志

本节我们将发散思维，从源码中解放出来，来思考一个更加具有实践意义的问题：如何改造 dispatch 方法，使得每次派发 action 时，都可以通过 console log 打印出相关信息，方便我们更加清晰地了解 store 当中 state 的每一步变更？

事实上，这正是 redux-logger 中间件所做的事情。从本节开始，我们将会一步步对中间件思想抽丝剥茧，这对于理解 Redux 中间件思想、设计 Redux 中间件非常关键，这也是体会 Redux 精妙设计的入门内容。

现在，我们就开始 dispatch 的改造计划。说是改造，其实更准确的表达是“覆盖”“重写”。dispatch 实质上就是一个函数，它负责调用 reducer 方法，依靠 reducer 的执行进行状态变更，接着依次执行各监听函数，目的是实现视图的更新。

这样，我们的入手点就应该在 dispatch 函数调用的前后，插入 console.log 状态的输出。Redux 的数据更新是同步进行的，当然最容易想到的办法是在业务代码中每一次 store.dispatch(action) 执行前后都手动进行记录。

```
console.log(action + "will dispatch");
console.log(store.getState());

store.dispatch(action);

console.log(action + "already dispatched");
console.log(store.getState());
```

但问题是我们不可能在代码中粗暴地加入零散琐碎的 console.log，而应通过扩展 dispatch 方法来实现。

为了改造 dispatch 函数，我们按照如下步骤进行操作。

(1) 创建一个名为 addLoggingToDispatch 的函数，用来生成取代“原始经典的 dispatch”的全新 dispatch 方法。

这个函数需要对 store 中的 dispatch 进行拦截,并记录原始的 dispatch 为 rawDispatch。同时, addLoggingToDispatch 函数应该在行为上同原始的 dispatch 保持一致,为此该函数返回一个新的函数,这个新的函数就是添加更新日志之后的全新 dispatch。所以,这个返回的函数自然要模仿 rawDispatch 的行为,将 action 作为参数。

```
const addLoggingToDispatch = (store) => {  
  const rawDispatch = store.dispatch;  
  // 返回的函数就是添加更新日志之后的全新 dispatch  
  return (action) => {  
    // ...  
  }  
}
```

(2)在返回的函数中,我们需要调用真实的 rawDispatch 方法,还原原始的 dispatch 的行为,同时在调用 rawDispatch 前后进行日志记录。需要注意的是,为了最大限度地还原原始的 dispatch 的行为,需要记录 dispatch 的返回值,并最终进行返回。

```
const addLoggingToDispatch = (store) => {  
  const rawDispatch = store.dispatch;  
  // 返回的函数就是添加更新日志之后的全新 dispatch  
  return (action) => {  
    // 按照 action 类型进行输出分组,保证在同一个 action 下拥有相同的日志 title  
    console.group(action.type);  
    // 打印更新前的 state  
    console.log('previous state', store.getState());  
    // 打印出当前 action  
    console.log('action', action);  
  
    // 调用原始的 dispatch 并记录返回值  
    const returnValue = rawDispatch(action);  
  
    // 打印更新后的 state  
    console.log('next state', store.getState());  
  
    console.group(action.type);  
  
    return returnValue;  
  }  
}
```

这样的处理很简单，操作也很灵活。

(3) 我们还可以使用颜色对日志输出进行美化，比如采用灰色修饰前一状态，`action` 本身使用蓝色，下一状态输出使用绿色。同时，对 `console.group` API 的兼容性进行判断和处理。

```
const addLoggingToDispatch = (store) => {
  const rawDispatch = store.dispatch;

  if (!console.group) {
    return rawDispatch;
  }

  // 返回的函数就是添加更新日志之后的全新 dispatch
  return (action) => {
    // 按照 action 类型进行输出分组，保证在同一个 action 下拥有相同的日志 title
    console.group(action.type);
    // 打印更新前的 state
    console.log('%c previous state', 'color: gray', store.getState());
    // 打印出当前 action
    console.log('%c action', 'color: blue', action);

    // 调用原始的 dispatch 并记录返回值
    const returnValue = rawDispatch(action);

    // 打印更新后的 state
    console.log('%c next state', 'color: green', store.getState());

    console.group(action.type);

    return returnValue;
  }
}
```

这样一个完美的增强型 `dispatch` 就开发出来了。将上述代码应用在 Redux 官方教程 `Todo App` 当中，我们便可以得到每一次 `action` 触发前后的记录信息，如图 4-1 所示。

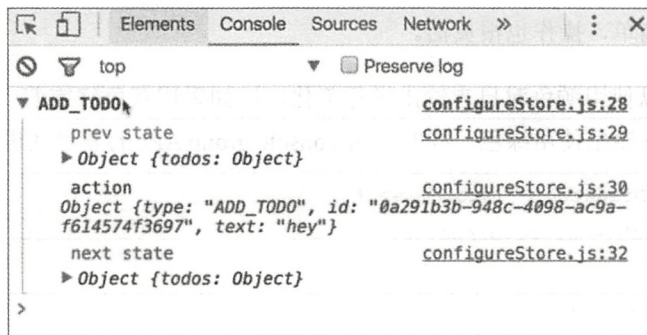


图4-1 logger中间件效果图

(4) 最后加入对开发环境和生产环境的判断。显然，我们只希望这种日志输出在开发阶段出现，以方便调试。如果是生产环境，则这些信息一般不适合输出。因此，可以在重写 `store.dispatch` 之前，先进行环境判断。

```
if (process.env.NODE_ENV !== 'production') {  
  const store.dispatch = addLoggingToDispatch(store);  
}
```

目前 `process.env.NODE_ENV` 已经成为前端工程化的一个使用规范，它的主要用途是在 Node.js 环境中执行脚本时，通过这个属性来区分不同环境（开发、生产、测试等）下的应用程序打包、构建、运行策略。

```
process.env.NODE_ENV === 'development'; // 或简写为 dev，意为开发环境  
process.env.NODE_ENV === 'production'; // 或简写为 prod，意为生产环境
```

至此，大功告成，我们对原始 `dispatch` 进行了功能性扩展。Redux 中间件多种多样，接下来我们继续了解通过对 `dispatch` 的扩展还能做些什么事情。

4.4 dispatch 的改造——识别 Promise

延续上一节的思路，我们希望对 `dispatch` 进行另一种改造：识别并处理 Promise。

在实际开发中，我们经常会遇到派发一个异步 action 的场景，这时就可以使用 `redux-thunk` 中间件来完成异步操作。其原理是令 `dispatch` 接收一个函数，在这个函数中进行异步操作。既然社区中有使 `dispatch` 接收一个函数的中间件 `redux-thunk`，那么也一定会有使 `dispatch` 接收一个 Promise 对象的中间件，它们都将帮助开发者完成异步 action 的处理。

我们知道 Promise 是用来解决异步问题的。同样地，如果 dispatch 能够接收一个 Promise 对象，我们就能处理 Redux 架构下的异步问题。具体思路是 dispatch 接收 Promise 对象，在这个 Promise 对象 resolve（状态发生改变）后，我们使用原始的 dispatch 进行 action 触发。这样的思路同 redux-thunk 中间件的思想并没有本质的区别。

按照上一节的实现，请参考如下代码。

```
const addPromiseSupportToDispatch = (store) => {
  const rawDispatch = store.dispatch;
  // 返回的函数就是添加更新日志之后的全新 dispatch
  return (action) => {

    // 对 action 进行判断，当是一个 Promise 对象时
    if (typeof action.then === 'function') {
      return action.then(rawDispatch);
    }

    return rawDispatch(action);
  }
}

if (process.env.NODE_ENV !== 'production') {
  const store.dispatch = addPromiseSupportToDispatch(store);
}
```

我们同样保留了对原始 store.dispatch 方法的引用（rawDispatch），以便稍后调用。同时返回了一个行为与原始的 store.dispatch 方法完全一致的增强型 dispatch 函数。

注意判断 Promise 的方式，这里使用了一个比较“投机取巧”的技巧：检查 action 参数是否具有 then 方法，且这个方法是函数类型。即判断 action 是否是一个 thenable 对象，如果是就会等待这个 Promise resolve，生成一个 JavaScript 对象，这个对象即为标准的 action，同时利用原始的 dispatch 对这个标准的 action 进行派发。如果 action 参数是一个 JavaScript 对象，那么就按照 rawDispatch(action) 方式进行操作。

另外，上面使用了如下代码来改写 dispatch 方法。

```
const store.dispatch = addLoggingToDispatch(store);
const store.dispatch = addPromiseSupportToDispatch(store);
```

请注意改写的顺序，如果将上面两行代码调换位置，就会出现 `action.type` 为 `undefined` 的情况，不符合我们的预期。

为了提升开发效率，在实际开发中也许需要经常改写 `dispatch`，那么 Redux 如何协调这些 `dispatch` 的改写呢？这就涉及中间件及中间件串联的知识了。其实每一次对 `dispatch` 的改写，都可以被封装成一个独立的中间件。请继续阅读下面章节的内容，了解 Redux 中间件组织和串联的奥秘。

4.5 糅合多种 dispatch

在前两节中，我们介绍了改造 `dispatch` 并增强其功能的两个例子，这就是 `redux-logger` 和 `redux-thunk` 这两个著名中间件的雏形，同时也奠定了理解 Redux 中间件的基础。

在开发时，也许需要更多地包装 `dispatch` 以实现更加完善的功能。比如，包装 `dispatch` 来打印日志，包装 `dispatch` 以支持 `Promise`，各种包装需要密切配合。下面汇总前两节的代码。

```
const addLoggingToDispatch = (store) => {
  const rawDispatch = store.dispatch;

  if (!console.group) {
    return rawDispatch;
  }

  return (action) => {
    console.group(action.type);
    console.log('%c previous state', 'color: gray', store.getState());
    console.log('%c action', 'color: blue', action);

    const returnValue = rawDispatch(action);

    console.log('%c next state', 'color: green', store.getState());
    console.group(action.type);
    return returnValue;
  }
}
```

```
const addPromiseSupportToDispatch = (store) => {  
  const rawDispatch = store.dispatch;  
  return (action) => {  
    if (typeof action.then === 'function') {  
      return action.then(rawDispatch);  
    }  
  
    return rawDispatch(action);  
  }  
}
```

为了使这两种包装方式同时运作,可以写一个用来初始化 store 的函数,以丰富 store.dispatch 的功能。

```
const configureStore = () => {  
  const store = createStore(App);  
  
  if (process.env.NODE_ENV !== 'production') {  
    store.dispatch = addLoggingToDispatch(store);  
  }  
  
  store.dispatch = addPromiseSupportToDispatch(store);  
  
  return store;  
}
```

这样便返回了一个包含有增强型 dispatch 的 store。仔细研究 configureStore 函数,我们发现 addPromiseSupportToDispatch 方法返回了一个符合正常用法的 dispatch,此时它支持 dispatch 参数是一个 Promise,它会等待 Promise resolve 后,利用 rawDispatch 再次进行 action 的派发。那么这个 rawDispatch 是最初、最原始的 dispatch 吗?在开发环境下,显然不是。因为在执行 store.dispatch = addPromiseSupportToDispatch(store) 之前,已经执行了 store.dispatch = addLoggingToDispatch(store)。

换句话说,在执行 addPromiseSupportToDispatch 时,store.dispatch 是上一个包装版本的 store.dispatch。明白了这层关系,也许我们会想到 rawDispatch 这样的命名并不十分合适。它本意是最原始的 store.dispatch,但是在代码执行时,每一个中间件所获得的 store.dispatch 都已经

被改造。接下来，我们就将其改名为 `next.addPromiseSupportToDispatch` 将是下一个包装 `dispatch` 的函数。

同理，`addLoggingToDispatch` 使用的参数也不一定就是最原始的 `dispatch`，实际上它可能位于包装链的任何一个位置。

```
const addLoggingToDispatch = (store) => {
  const next = store.dispatch;

  if (!console.group) {
    return next;
  }
  return (action) => {
    // ...

    const returnValue = next(action);

    // ...
    return returnValue;
  }
}

const addPromiseSupportToDispatch = (store) => {
  const next = store.dispatch;
  return (action) => {
    if (typeof action.then === 'function') {
      return action.then(next);
    }
    return next(action);
  }
}
```

理解了这些内容，我们有理由判断出，采用上述方式直接零散地修改公共的 API 接口，然后代替自定义函数，并不是一种很好的做法。在实际应用开发中，如果由开发者手动处理各种改写逻辑，也会非常混乱。

为了规避这样的反模式，我们可以将这种包装过程收敛——声明一个数组，即中间件数组，

它的每一项就是一个中间件，然后统一根据中间件来增强 `dispatch`。

说了这么多，那么中间件到底是什么呢？其实就是上面提到的 `addLoggingToDispatch`、`addPromiseSupportToDispatch`。

所以，Redux 的核心思想就是将 `dispatch` 增强改造的函数（中间件）先存起来，然后提供给 Redux，Redux 负责依次执行。这样每一个中间件都对 `dispatch` 依次进行改造，并将改造后的 `dispatch` 即 `next` 向下传递，即将控制权转移给下一个中间件，完成进一步的增强。

具体体现在代码上就是：

```
const configureStore = () => {
  const store = createStore(App);
  const middlewares = [];

  if (process.env.NODE_ENV !== 'production') {
    middlewares.push(addLoggingToDispatch);
  }
  middlewares.push(addPromiseSupportToDispatch);

  wrapDispatchWithMiddlewares(store, middlewares);

  return store;
}
```

在以上代码中，并不直接执行 `addPromiseSupportToDispatch` 和 `addLoggingToDispatch`，而是将其 `push` 到 `middlewares` 数组，然后统一执行。这里所说的统一执行，对应的就是代码中的 `wrapDispatchWithMiddlewares(store, middlewares)`。

接下来，就涉及如何编写 `wrapDispatchWithMiddlewares` 函数了。因为中间件统一执行的过程一定依赖 `middlewares` 数组和最初纯净的 `store.dispatch`，所以这两项将会作为参数传入 `wrapDispatchWithMiddlewares` 函数当中。由此可知，`wrapDispatchWithMiddlewares` 需要做的就是逐一执行中间件，为此我们使用数组的 `forEach` 方法。

```
const wrapDispatchWithMiddlewares = (store, middlewares) => {
  middlewares.forEach(middleware =>
    store.dispatch = middleware(store)(store.dispatch);
  )
}
```

同时，也需要改写中间件，因为之前的设计是中间件直接读取一个增强后的 `dispatch`，而在连接各个中间件时，需要返回一个返回函数的函数。这样做的意义在于：各个中间件不必再到 `store` 中读取 `dispatch`，而是将增强的 `dispatch` 作为参数进行传递和连接，进而层层递进完成控制权的转移。

改写后的两个中间件如下：

```
const promise = (store) => (next) => (action) => {
  if (typeof action.then === 'function') {
    return action.then(next);
  }
  return next(action);
}

const logger = (store) => (next) => {

  if (!console.group) {
    return next;
  }

  return (action) => {
    // ...

    const returnValue = next(action);

    // ...
    return returnValue;
  }
}
```

注意：为了更能表达出中间件的作用，这里将 `addLoggingToDispatch` 改名为 `logger`，将 `addPromiseSupportToDispatch` 改名为 `promise`。

最后，`middlewares` 数组执行的顺序与我们预期执行的顺序相反，为此，需要做出反转修复。

```
const wrapDispatchWithMiddlewares = (store, middlewares) => {
  middlewares.slice().reverse().forEach(middleware =>
```



```

    store.dispatch = middleware(store)(store.dispatch);
  )
}

```

这样一来, `logger` 中间件出现在数组的末尾, 所以它的 `next` 参数就是最原始的 `store.dispatch`; 而 `promise` 中间件的 `next` 参数就是经过 `logger` 包装后的 `dispatch`。

4.6 Redux 源码探索——中间件的秘密

其实中间件可以为多种目的服务, 比如 `logger` 中间件用于打印调试信息; `promise` 中间件用于增强 `dispatch`, 使其接收一个 `Promise` 对象等。

为了使多个中间件共同协作, 我们创建了 `middlewares` 数组, 依据数组的每一项将 `dispatch` 层层包裹进行增强。

中间件名称的由来, 其实就是因为它使得 `action` 在到达 `reducer` 之前, 增加了一个中间环节, 我们可以充分利用此环节来完成日志输出、数据分析、错误处理、异步流程等。

相信不止一个人认为, 中间件是 `Redux` 最出彩的设计之一, 但是很多开发者尝试阅读 `Redux` 关于中间件的源码时会感到头疼。其实如果要真正理解中间件的源码, 则不妨尝试从已有的中间件入手, 自己再动手开发一个中间件, 循序渐进是非常合适的学习路径。讲到这里, 也许你已经发现, 中间件虽然嵌套了多层函数, 各层函数又返回函数, 但是很多中间件的实现仅仅就 10 行代码而已, 便发挥了巨大威力。

4.6.1 源码剖析

通过前面两节的介绍, 相信读者已经掌握了中间件的核心内容。本节我们将直接深入源码一探究竟。现在我们有必要回顾一下在进行 `Redux` 开发时应用中间件的情况。

`Redux` 本身暴露了一个 `applyMiddleware` 的接口, 我们只需要按照下面方式引入即可。

```
import { createStore, applyMiddleware } from 'redux';
```

同时, 对于上述两个中间件, 在 `NPM` 中也各有实现。我们暂且使用现成的 `redux-promise` 和 `redux-logger` 库。

```
import promise from 'redux-promise';
import createLogger from 'redux-logger';
```

这两个开源库更加完备，而且增加了很多可以自定义的内容。

```
const configureStore = () => {
  const middlewares = [];

  if (process.env.NODE_ENV !== 'production') {
    middlewares.push(createLogger());
  }

  return createStore(
    reducer,
    applyMiddleware(...middlewares)
  )
}
```

`applyMiddleware` 返回的内容我们称为 `enhancer`，这是 `createStore` 方法的最后一个参数，并且是可选的。另外，如果想初始化一些数据，则可以这样写：

```
let persistedState = { ... };
return createStore(
  reducer,
  persistedState,
  applyMiddleware(...middlewares)
)
```

在 `Redux` 源码中，涉及中间件的脚本有 `applyMiddleware.js`、`createStore.js`、`compose.js`。那在 `createStore` 方法中，`applyMiddleware(...middlewares)` 会发生什么事呢？我们找到 `createStore.js` 文件的源码部分：

```
export default function createStore (reducer, preloadedState, enhancer) {
  // ...

  if (typeof enhancer !== 'undefined') {
    if (typeof enhancer !== 'function') {
      throw new Error('...')
    }

    return enhancer(createStore)(reducer, preloadedState)
  }
  // ...
}
```

顾名思义，`applyMiddleware` 就是对各个需要应用的中间件进行糅合，并作为 `createStore` 的第二个或者第三个参数传入，用于增强 `store`。源码如下：

```
export default function applyMiddleware (...middlewares) {
  return (next) =>
    (reducer, initialState) => {
      var store = next(reducer, initialState);
      var dispatch = store.dispatch;
      var chain = [];

      var middlewareAPI = {
        getState: store.getState,
        dispatch: (action) => dispatch(action)
      };
      chain = middlewares.map(middleware => middleware(middlewareAPI));
      dispatch = compose(...chain, store.dispatch);

      return {
        ...store,
        dispatch
      };
    }
}
```

在这几行代码中，应用了大量的函数式编程概念，如高阶函数、函数组合、柯里化等。下面我们进行拆解，以方便理解。

```
export default function applyMiddleware (...middlewares);
```

这里使用了扩展运算符，使得 `applyMiddleware` 可以接收任意个数的中间件。接下来，`applyMiddleware` 会返回一个函数，这个函数接收了一个 `next` 参数：

```
return (next) => (reducer, initialState) => {...}
```

对应于 `createStore.js` 代码，`applyMiddleware` 作为一个三级柯里化的函数，它的执行相当于：

```
applyMiddleware(...middlewares)(createStore)(reducer, initialState)
```

这样做的目的是借用原始的 `createStore` 方法，创建一个新的增强版 `store`，具体看其内部实现：

```
var store = next(reducer, initialState);
var dispatch = store.dispatch;
var chain = [];
```

这里记录了原始的 `store` 和 `store.dispatch` 方法，并准备一个 `chain` 数组。

```
var middlewareAPI = {
  getState: store.getState,
  dispatch: (action) => dispatch(action)
};
chain = middlewares.map(middleware => middleware(middlewareAPI));
```

`middlewareAPI` 是第三方中间件需要使用的参数，即原始的 `store.getState` 和 `dispatch` 方法，这些参数在中间件中是否会全部应用到，自然要看每个中间件的应用场景和需求。实际上，可以形象地将 `middlewareAPI` 理解为 `Redux` 为中间件准备的“武器”或者“后门”，用来让社区中间件发挥效力。中间件也是一个高度柯里化的函数，它接收 `middlewareAPI` 参数后的第一层返回结果，并存储到 `chain` 数组中。

可以想象，`chain` 数组中的每一项都是对原始 `dispatch` 的增强，并进行控制权转移。所以就有了 `dispatch = compose(...chain, store.dispatch)`。

这里的 `dispatch` 函数就是增强后的 `dispatch`。因此，`compose` 方法接收了 `chain` 数组和原始的 `store.dispatch` 方法。

```
export default function compose (...funcs) {
  if (funcs.length === 0) {
    return arg => arg
  }

  if (funcs.length === 1) {
    return funcs[0]
  }

  return funcs.reduce((a, b) => (...args) => a(b(...args)))
}
```

实际上，`compose` 方法就是把多个中间件串联起来，就像：

```
middlewareA(middlewareB(middlewareC(store.dispatch)))
```

这是只有三个中间件 `middlewareA`、`middlewareB` 和 `middlewareC` 的情况，关于有更多中间件的情况，依此类推。

4.6.2 写一个中间件的套路

了解了中间件的工作原理后，自己开发一个中间件也非常简单。事实上，得益于 Redux 设计思想，所有中间件的编写都是有“固定套路”或“模式”的。

```
const customMiddleware = store => next => action => {  
  // ...  
}
```

我们可以应用中间件编写模式，完成一个全新的中间件的开发。设想这样一个场景：应用存在多套主题皮肤可供用户选择切换。这些皮肤在一定时间内往往都是有固定样式的，在初始化整个应用时，使用一套默认的主题皮肤。在用户切换主题的情况下，我们希望用户离开应用后，下次再访问此应用时，仍然可以直接切入上一次切换后的主题，而不是默认主题。

切换一套主题皮肤的 `action` 如下：

```
store.dispatch({  
  type: 'CHANGE_THEME',  
  payload: 'light'  
});
```

那么，可以这样定义一个 `CHANGE_THEME` 中间件：

```
const CHANGE_THEME = store => next => action => {  
  // 拦截目标 action  
  if (action.type === 'CHANGE_THEME') {  
    if (localStorage.getItem('theme') !== action.payload) {  
      localStorage.setItem('theme', action.payload)  
    }  
  }  
  
  return next(action);  
}
```



在应用这个中间件的情况下，业务的初始化脚本如下：

```
store.dispatch({
  type: 'CHANGE_THEME',
  payload: localStorage.getItem('theme') || 'dark'
});
```

设想用户第一次进入应用时，因为无法通过 `localStorage.getItem("theme")` 获取先前的主题皮肤，所以 `payload` 为默认的 `'dark'`。当派发 `'CHANGE_THEME'` action 之后，被 `CHANGE_THEME` 中间件拦截，并设置相应的主题皮肤。

下一次用户访问时，会自动派发：

```
{
  type: 'CHANGE_THEME',
  payload: 'dark'
}
```

当切换为蓝色主题皮肤时，派发：

```
{
  type: 'CHANGE_THEME',
  payload: 'blue'
}
```

都会被 `localStorage` 存储，以便下一次访问时再次应用。

这就是通过设计一个中间件，拦截用户对主题皮肤的更改，并存入 `localStorage` 的典型应用。

4.7 再谈 Redux 设计思想

经过前面几节的介绍，相信大家对 Redux 的内部工作机制、实现原理都有了更深入的理解。本节我们将站在更高的角度，从整体上来分析 Redux 设计思想。

Redux 一经推出，便获得了潮水般的好评。就目前来看，Redux 搭配 React 的技术栈已经非常流行，就连 Redux 的作者 Dan Abramov 也没有想到它能这么迅速地风靡前端开发，并建立起强大而有活力的社区。那么到底是什么让 Redux 如此成功呢？



4.7.1 Redux 的特性和限制

当我们在调研一个第三方类库或者框架的时候，首先会关注它的特性和 API，进而判断该类库或框架应用是否符合自己的需求。但是当我们把目光投向 Redux，了解其内部实现时，就会发现它并没有带来任何新特性，其本质上就是一个事件发布订阅系统。

```
let listeners = [];  
let state =;  
  
function dispatch (action) {  
  state = reducer(state, action);  
  listeners.slice().forEach(i => i());  
  return action;  
}  
  
function getState() {  
  return state;  
}  
  
function subscribe (listener) {  
  listeners.push(listener);  
  return function () {  
    listeners = listeners.filter(i => i !== listener);  
  }  
}
```

对于这样的事件发布订阅系统，前端开发者一定不陌生。

因此，开发者选择 Redux，并不是因为它带来了新特性。但是作为技术选型人员，热衷于 Redux 的原因是什么呢？

现在我们来思考这样一个问题：任何新特性或者创新思路，在带来功能强大且丰富的 API 的同时，也带来了一些潜在的看不见的影响。那么，新特性和 API 带来的“看不见”的一面是什么呢？那就是代码实现方式上的限制和制约。

我们来看看 Redux 带来的限制。



- 单一 store，或单一状态。
- action 描述。
- reducer 处理更新。

首先，Redux 限制了应用状态树就是一个对象或者 JavaScript 数组，使用者不能自定义数据模型。

其次，如果想改变一些东西，比如更新应用的状态，Redux 不提供任何 setter 方法，使用者唯一能做的就是派发一个 action 来描述状态的更新。这个 action 也需要是一个 JavaScript 对象。

最后，为了应用这些更新，使用者需要专门编写一些 reducer 函数，并且这些函数要保持纯净。

这些限制也有很多益处，比如给调试带来巨大的便利性，具体体现为：

- 可以记录 action 和状态。
- 查询不合理的状态。
- 检查审核 action。
- 对症下药，修复 reducer。
- 编写测试用例。

我们可以记录每一个 action 和状态，当程序出现问题时，首先查询究竟是哪一个状态发生错误，然后回溯，看前一个触发错误的 action 是否准确。因为 action 都是 JavaScript 对象，完全可读，所以我们完全有能力识别出 action 的正确性。比如有如下这样的 action：

```
{
  type: 'SELECT_USER',
  data: {
    userId: undefined
  }
}
```

因为 action 的负载信息 userId 出现了 undefined 的情况，所以有很大的可能性是这个 action 出现了错误。为此，我们可以马上定位问题所在，来核实正确的 userId 值。



如果这个 action 是正确的，那么我们就可以找到相应的 reducer，查看处理'SELECT_USER'的 reducer 是否做出了错误的状态更新。

这样的流程也非常方便编写测试用例。我们无须一次次刷新页面，无须每次都进行页面渲染，需要做的就是编写测试函数，观察对应的输入是否返回了预期的状态。

另外，Redux 带来的便利性还体现在“一切都是数据”上，这就给我们带来了持久性、同构渲染、记录用户 session、优化共享、协同开发等便利。在持久性方面，数据采用 JavaScript 对象来存储，可以方便地使用 `Json.parse` 等相关序列化方法，可以在任何需要数据的地方进行加载。同时，对于同构渲染，我们也可以在服务端就进行状态的设定，完成服务端直出。当页面出现错误时，我们可以用典型的 `try...catch` 捕获错误，并记录当前的状态以及 action，这对于生成页面错误报告也是非常有意义的。对于优化共享，我们可以设想这样的场景：在页面中，一个用户可以点击关注按钮去关注另外一个用户。在点击关注之后，可以在发送网络请求之前便对 state 进行最初的更改，同时把相关的 action 记录在一个队列中，稍后再发送关注的请求。如果请求最终失败，则返回相应的状态，并提示用户。这样的处理过程由数据驱动页面状态，一切转化都基于数据，因此一切就都在开发者的掌控之中了。Redux 的设计非常有利于团队协作开发，以减少开发冲突等问题。

4.7.2 Redux 生态

说到 Redux，就不得不谈它的生态和社区。在这方面，我们可以归纳出几个关键概念：

- reducer
- selector
- 中间件
- 增强器

reducer 是可以组合的，也就是说，一个 reducer 可以调用另一个 reducer。在 Redux 官方的 Todo App 示例中，我们可以找到相关的代码：

```
// Slice reducer
const todos = (state = [], action) => {
  switch (action.type) {
```



```
    case 'ADD_TODO':
      return [...state, todo(undefined, action)]
      // ...
    default:
      return state;
  }
}
```

这里的 todos reducer 在处理 'ADD_TODO' action 时，又调用了 todos reducer。

另外，selector 这个概念并不是 Redux 提出的，但是在真正的开发中，它往往成为最佳实践。一个 selector 就是一个函数，其基本结构如下：

```
(state, ...args) => derivation;
```

它接收当前 state 和一个可选参数，根据参数计算出 UI 真正需要的数据信息。同样，在 Redux 官方的 Todo App 示例中，也有相应的代码：

```
const getVisibleTodos = (state, filter) => {
  switch (filter) {
    case 'all':
      return state;
    case 'completed':
      return state.filter(t => t.completed);
  }
}
```

关于 reducer 的相关概念，这里再补充一个高阶 reducer：

```
(reducer, ...args) => reducer;
```

所谓高阶 reducer，就是指一个函数接收一个现有的 reducer 和参数，并返回一个新的 reducer。这么解释也很抽象，我们来看一个示例。

在现有的已开发完成的页面应用中，如果想接入类似于“撤销”的功能，即用户输入或者编辑的信息可以随时撤销，那么对于这样的设计，为了更好地实现复用，可以采用高阶 reducer 来完成。主要思路是接收已有的 reducer，并补充加入处理 'UNDO' action 的逻辑，如果命中这个 action，就从记录信息中返回前一个 state，但前提是需要对每一个 state 进行记录。

```
const undoable = (reducer) => {
  const initialState = {
```



```
    past: [],
    present: reducer(undefined, {})
  }
  return (state = initialState, action) => {
    const {past, present} = state;
    if (action.type === 'UNDO') {
      return {
        past: past.slice(0, -1),
        present: past[past.length - 1]
      }
    }
    return {
      past: [...past, present],
      present: reducer(present, action)
    }
  }
}
```

这么一个 undoable 高阶 reducer，是可以复用的，只需植入到任意想要接入“撤销”功能的应用中即可。

另外，对高阶 reducer 的应用还体现在很多第三方库中，比如由 mattkrick 编写的 `redux-optimistic-ui`、由 davidkpiano 编写的 `redux-redux-form` 等。

正是 Redux 灵活的设计，才造就了第三方社区的繁荣。说到这里，更具有代表性的概念就是中间件了。所有合格的中间件都可以被发布到 NPM 当中，激发 Redux 社区的无限活力和可能。关于增强器的概念，也不同于传统的基于类的开发语言，它代表了典型的函数式思想，具备组合的强大功能。

Redux 设计还带来了可迁移的灵活性。比如，在已经应用 Redux 并编写了相应的 reducer 函数的设计中，如果有一天不想使用 Redux 了，Redux 也为我们设计了后退“出路”。因为 reducer 只是处理 state 的函数，因此脱离 Redux 它依然可用，比如在 React 组件中，完全可以使用如下代码来进行页面数据或组件状态的更新。

```
this.setState(prevState => reducer(prevState, action));
```




4.8 react-redux 究竟是什么

通过前面章节的学习，我们了解到 React 和 Redux 本身并没有什么关联，React 负责视图层的构建，Redux 负责数据流的处理。从这方面来看，React 和其他类库搭配使用也是可以的，比如 MobX；Redux 也可以和任意具有视图构建能力的类库组合使用，比如 jQuery，甚至单独使用 Redux 来做事件订阅发布系统，也未尝不可。

在 React 和 Redux 结合的情况下，如果每次在 React 组件中获取状态的最新变化都使用 `store.getState()`，显然是极其烦琐且不清晰的设计。而 react-redux 的出现，使一切变得清晰而简单。

简单的 react-redux 使用模式是将所有的业务组件嵌套在由 react-redux 提供的 Provider 组件当中，并将所生成的 store 设置为 Provider 组件的参数，Provider 组件便感知到 store。

```
import React from 'react'
import { render } from 'react-dom'
import { Provider } from 'react-redux'
import { createStore } from 'redux'
import App from '../components/App'

let store = createStore(todoApp);

render(
  <Provider store={store}>
    <App />
  </Provider>,
  document.getElementById('root')
)
```

react-redux 的核心功能只有两个：Provider 组件和 connect 方法。下面我们来逐一分析。

4.8.1 Provider 组件

react-redux 提供了 Provider 组件，它的作用是让业务组件获得 store 的信息。那么，Provider 组件作为 App 组件的父组件，是如何将 store 传递给 App 组件的呢？

我们知道 React 父子组件之间的通信，一般是通过使用 props 和基于 props 的回调来实现的，



但是在上面的代码中，很明显并没有这样做。那么 store 信息是如何传递给容器组件的呢？秘密就在于 React 的高级特性——context。

context 用来使 React 子孙组件可以直接“越级”获取父组件的信息，这样就不再需要一层一层通过 props 向下传递了。我们可以直接参考 react-redux 源码中的 react-redux/src/components/Provider.js 文件（下面只贴出核心逻辑代码）。

```
class Provider extends Component {
  getChildContext() {
    return { [storeKey]: this[storeKey], [subscriptionKey]: null }
  }

  constructor (props, context) {
    super(props, context)
    this[storeKey] = props.store;
  }

  render() {
    return Children.only(this.props.children)
  }
}
```

其中 `this.props.children` 表示组件的所有子节点，`Children.only` 保证 `children` 中仅有一个子级，否则抛出异常。这也就说明了 `react-redux` 要求 `App` 组件只能作为唯一的最顶层容器组件出现（组件命名可由开发者自行确定），而不能平行放置其他节点的原因。

在 `Provider` 组件中，`constructor` 方法用于初始化获取 `props` 中的 `store` 对象，同时通过 `getChildContext` 方法将外部的 `store` 对象放入 `context` 中。这样对子组件使用 `connect` 方法，就可以直接访问到 `context` 对象中的 `store`。

提到 `connect`，我们来看一下它的实现原理。

4.8.2 connect 方法

`Provider` 组件提供了直接访问 `store` 的基础，`connect` 方法真正连接了 `Redux store` 和 `React` 组件。`connect` 方法通过传入的 `mapStateToProps`、`mapDispatchToProps`、`mergeProps`、`options` 等

参数，计算出应该传递给 React 组件哪些属性和信息。

我们可以参考 react-redux 源码中 react-redux/src/connect 下的代码，合并并提取核心逻辑，并且归纳出该方法的执行步骤如下：

- 通过 context 获取 Provider 的 store，因此它具有了访问 store.state 的能力。
- connect 方法返回一个函数，该函数接收外部传入的业务组件，并且返回一个注入了 store 相关信息的 React 组件。
- 这个返回的 React 组件重新渲染外部传入的原业务组件，并把 connect 中传入的 mapStateToProps、mapDispatchToProps 等与组件中原有的 props 合并。

将这些代码高度“浓缩”，其实就是：

```
connect(  
  mapStateToProps, mapDispatchToProps, mergeProps, options: object  
) => ( component ) => connectComponent
```

这里对 connect 做了一个大方向上的指引，读者只需明白 Provider 就是一个组件，其核心目标是将 props.store 放到 context 中；connect 作为一个柯里化的高阶函数，可以根据一级参数计算筛选出 store 信息，根据二级参数 component 返回一个高阶组件 connectComponent 即可。

react-redux 设计示意图如图 4-2 所示。

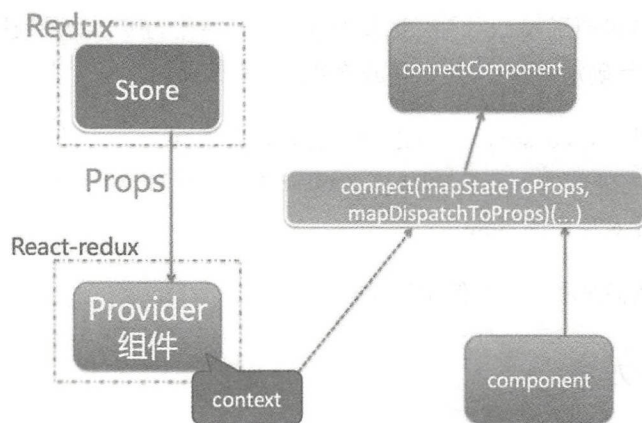


图4-2 react-redux设计示意图

上面提到的是 `react-redux` 最核心的思想，同时也充分体现了 `React` 组件的灵活性、可扩展性。

4.9 本章小结

本章内容较为深入，剖析了 `Redux` 源码及 `react-redux` 的本质。相信通过本章的学习，读者对 `Redux` 会有一个更高层面的认知。本章中的某些内容较为晦涩，函数式风格的代码也较为抽象，需要读者开动脑筋，并结合实践进行体会。

停留在技术的使用层面不是最终目的，深入其设计和实现底层才能全面掌握技术。在进行技术选型、技术方案制定时，“拍脑袋”无法解决问题，每一位开发者都应该具有探索精神。

第 5 章

揭秘 React 同构应用

本章我们将深入介绍 React 技术生态的服务端渲染以及开发同构应用。前后端分工，不仅是开发流程上的重要环节，而且是一个站点或应用在设计上的体现。长期以来，前后端界限的划分标准在不断更迭演进，它将是未来持续讨论的一个话题。

React 和 Redux 为服务端渲染提供了优良特性，借着 Node.js 发展的势头，同构应用变得越来越普遍。为了方便高效地开发同构应用，社区中也出现了类似于 Next.js 这样优秀的框架。作为开发者，即使所采用的技术架构并不是服务端渲染的同构设计，也很有必要对同构设计进行了解并掌握其原理。

5.1 前后端架构设计和服务端渲染概念

服务端渲染或直出的概念越来越流行。在了解如何基于 React 实现服务端渲染之前，我们有必要在架构层面对服务端渲染的“前世今生”做一个整体了解。比如需要弄清楚为什么会出现这样一个概念；这个概念落地之后，能解决什么问题；服务端渲染和其他方式对比有何利弊，等等。

5.1.1 前后端配合技术的演进

让我们从前后端分离的技术“纷争”说起。

早期的 Web 开发，架构设计简单、直接，具体来讲，就是页面由 JSP、PHP 等工程师在服务端生成，浏览器只负责展现。那时候，前端工程师只需要给静态页面添加一些动态交互效果，很少会涉及数据逻辑等；而后端工程师负责页面内容，即当用户请求页面时，后端进行处理并

返回完整的静态页面。一般这些过程会依靠模板引擎来完成。因此，在那个时候，甚至没有独立的前端工程师职位。即使有的话，这种做法的缺点也很明显，比如前后端分工职责不清，模板的编写由谁来完成？如果由前端人员来开发模板，那么前端将会极度依赖后端环境，难以实现开发效率的最大化，同时关于数据格式的沟通成本也相对较高。另外，这样的架构模式对于前端技术的发挥和利用浏览器能力的空间是非常有限的。

随着前端技术的飞速发展，尤其是 AJAX 和 Node.js 等技术的出现，一种前后端分离的架构模式应运而生。在这种模式下，前后端分工变得非常清晰，两端的关键协作点是 AJAX 接口。下面我们以用户访问页面为例来一步步了解这种模式，如图 5-1 所示。

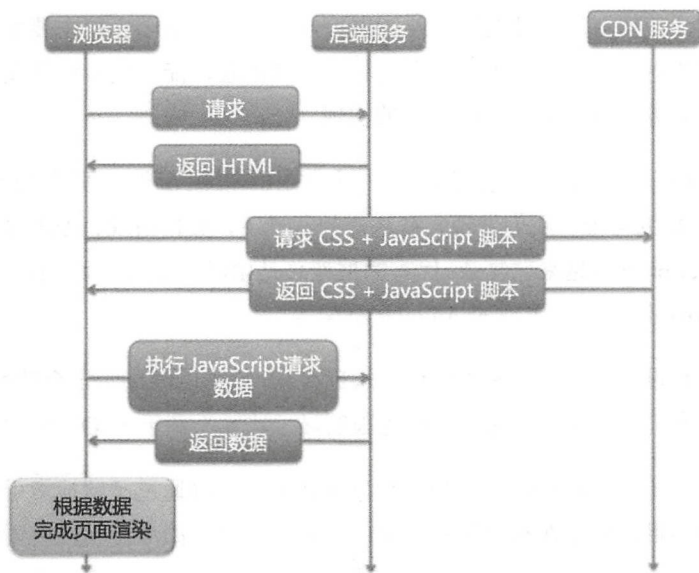


图5-1 非服务端渲染访问流程图

在这种构架设计下，对页面的请求处理分为以下几个步骤。

- (1) 浏览器请求页面。
- (2) 服务端返回“不包含页面内容”的 HTML 文件。
- (3) 浏览器加载静态页面，解析 HTML 文件。

(4) 在 HTML 文件中遇见所需的 CSS 资源（一般 CSS 文件放于头部），进行请求并拉取资源，这里假设静态资源存储在 CDN 服务器上。

(5) 在 HTML 文件中遇见所需的 JavaScript 资源（一般 JavaScript 文件放于尾部），进行请求并拉取脚本（在图 5-1 中将对 CSS 资源和 JavaScript 资源的请求合并并在同一个流程当中，实际上对不同资源的请求有先后顺序，会受到浏览器最大并行数的限制）。

(6) 当 JavaScript 文件加载完成后，执行 JavaScript 脚本。

(7) 在 JavaScript 脚本中包含了对页面所需数据的异步请求，此时通过 AJAX 来获取数据。

(8) 数据请求成功后，由 JavaScript 脚本完成对数据的处理，并根据数据渲染到页面进行展现。

这样的架构设计使得前后端开发可以并行进行，职责清晰。前端工作主要集中在浏览器端，前端开发只需要完成对测试数据的模拟，环境相对容易配置，能做到本地开发，脱离后端的支持；而后端专注于业务逻辑，负责 API 接口的实现。

这样的架构设计一经推出，便得到了广泛应用。它也使得单页面应用（Single Page Application, SPA）成为可能。我们发现，在这样的背景下，数据请求、处理等复杂逻辑被移植到浏览器端，JavaScript 脚本越来越复杂。优秀的框架总是由时势造就，涌现出一系列解决方案，比如早期的 Backbone 以及后来出现的 Angular 框架等。

任何技术架构和设计都不可能脱离时代而永远存在，技术的演进一定会随着发展愈演愈烈。这种架构模式在提升开发效率的同时，短板也很明显，比如不利于 SEO 和存在性能问题等。

SEO（Search Engine Optimization，搜索引擎优化）是一种通过了解搜索引擎的运作规则来调整网站，以及提高目的网站在有关搜索引擎内排名的方式。如今，网民主要通过搜索引擎在网上查找信息和资源。SEO 做得好，能够直接提升网站在搜索引擎搜索结果中的展现排名，更有利于页面的曝光。

可是采用前后端分离的方式，由于页面的数据内容主要由 JavaScript 脚本动态生成，因此非常不利于搜索引擎获取该页面的信息，影响该页面的 SEO。虽然网页爬虫可能会分析动态请求，但是这样的技术并不成熟。

另外，在这种架构设计下，我们会发现来自浏览器端的请求增多了，体现在用户体验上，就是用户必须等待 JavaScript 脚本加载完成，且真正执行时才会发起数据请求。接下来，等待数据成功返回后，脚本完成页面内容渲染，用户才可以得到最终页面。这样做直接降低了页面首屏展现的时间，特别是在移动互联网环境下，对首屏加载性能的影响很大。

为了解决上述问题，所谓的服务端渲染就粉墨登场了。

5.1.2 技术历史总是惊人的相似

关于服务端渲染，各方的定义和用词都不尽相同，实际上它也依赖不同开发者的理解，但是其原理是相似的。

服务端渲染技术会把数据请求过程放在服务端，相对于前后端分离的方式，获取数据更加提前，页面模板结合数据的渲染处理也在服务端完成。结合 React 技术，基本的组件拼接在服务端完成，并最终输出相对完整的 HTML 返回给浏览器端。接下来，进一步的组件渲染将在浏览器端完成。具体流程如图 5-2 所示。

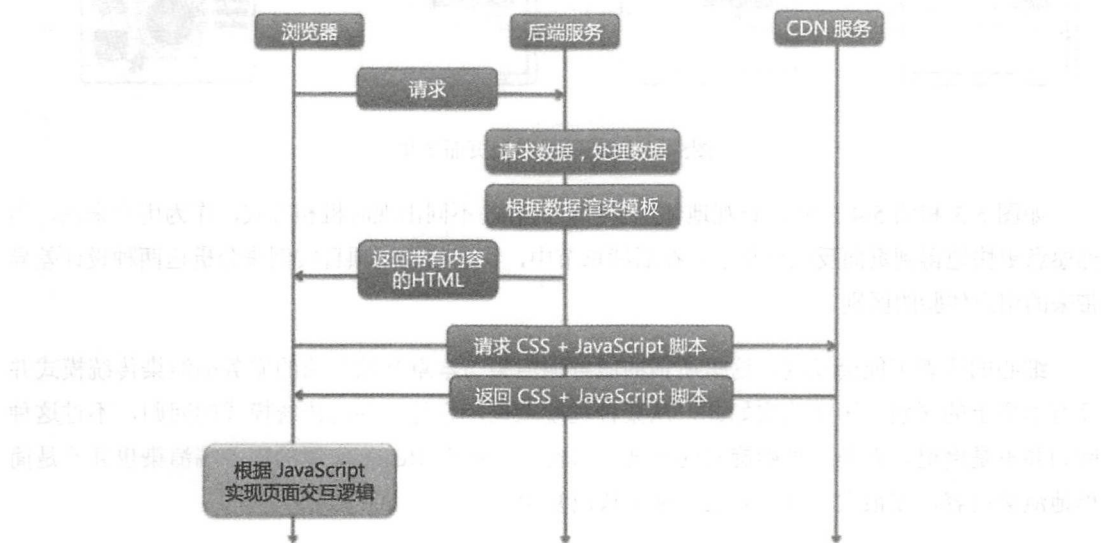


图 5-2 服务端渲染访问流程图

这样做的好处非常明显：当浏览器初次请求页面后，用户第一次拿到的 HTML 文档已经进行了初步内容渲染，这样必然更有利于 SEO 优化，也解决了首屏的性能问题。

我们发现总的请求数并没有改变，而是把浏览器的一部分数据请求移到了服务端。事实上，在服务端进行数据拉取的成本要远远小于浏览器端，而且传输更加高效，这也是性能提升的关键之处。

图 5-3 和图 5-4 展示了在两种不同的实现方式下，对首屏体验的不同影响。

服务端接到请求后，准备数据，渲染模板，将会返回带有内容的HTML文档



浏览器渲染HTML，此时页面已具备基本内容，同时请求静态资源



浏览器加载并执行JavaScript脚本

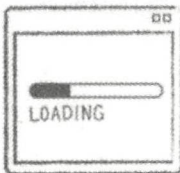


JavaScript脚本运行完毕，此时页面可交互

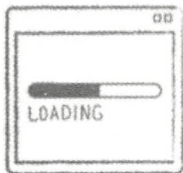


图5-3 服务端渲染页面效果

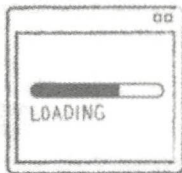
浏览器发出请求



服务端返回无数据HTML，浏览器请求并下载JavaScript



浏览器执行JavaScript脚本



JavaScript脚本执行完毕，页面内容被渲染，且可交互



图5-4 浏览器端渲染页面效果

如图 5-3 和图 5-4 所示，直观地体现了首屏内容的不同出现时机和方式，作为用户来说，当然愿意更快地得到页面反馈信息了。在后续章节中，我们将通过项目实例来分析这两种设计差异带来的用户体验的区别。

细心的读者可能会发现，这里所谓的服务端渲染与本章开始介绍的服务端渲染传统模式并没有本质上的区别。事实确实如此，从某种程度上说，它是一种向传统模式的回归，不过这种回归并不是倒退，而是一种螺旋式的发展。事实上，依靠 React 实现的服务端渲染也并不是简单地渲染内容，在很大程度上它还实现了代码复用。

5.2 同构应用

本节我们将“服务端渲染”一词替换为“同构”。其实，这两个词的背景和所表达的意义大体相同，但又有一定的差别：服务端渲染主要侧重架构层面的实现，而同构更侧重代码复用。

任何一种架构模式都是以服务业务需求为前提、以技术时代发展为背景的。它们各有利弊，具体采用哪一种模式，需要开发者深思熟虑，结合自身的实际情况进行选择。本节我们将列举构建同构应用的优缺点。

5.2.1 什么是同构

随着 Node.js 的异军突起，前后端开发有了归一化编程语言的基础土壤，页面模版、第三方依赖机制等都有实现前后端统一的契机。React 率先引领了这种潮流，同构的概念也因此得以更广泛的传播。

需要读者明白的是，同构应用并不是不需要浏览器端渲染内容，而是使服务端和浏览器端渲染达到一种平衡。那么，怎么理解这种平衡呢？

在服务器上生成渲染内容，让用户尽早看到有信息的页面。一个完整的应用除包括纯粹的静态内容以外，还包括各种事件响应、用户交互等。这就意味着在浏览器端一定还要执行 JavaScript 脚本，以完成绑定事件、处理异步交互等工作。

从性能及用户体验上来看，服务端渲染应该表达出页面最主要、最核心、最基本的信息；而浏览器端则需要针对交互完成进一步的页面渲染、事件绑定等增强功能。所谓同构，就是指前后端共用一套代码或逻辑，而在这套代码或逻辑中，理想的状况是在浏览器端进一步渲染的过程中，判断已有的 DOM 结构和即将渲染出的结构是否相同，若相同，则不重新渲染 DOM 结构，只需要进行事件绑定即可。

从这个维度上讲，同构和服务端渲染又有所区别，同构更像是服务端渲染和浏览器端渲染的交集，它弥补了服务端和浏览器端的差异，从而使得同一套代码或逻辑得以统一运行。同构的核心是“同一套代码”，这是脱离于两端角度的另一个维度。

5.2.2 同构的优势和劣势

同构的优势如下：

- 更好的性能。这里的性能主要指渲染更加迅速、首屏展现的时间更快、文件更少，以及文件体积更小。
- SEO 优化支持。服务端接收到请求后，会返回一个相对完整、包含了初始内容的 HTML 文档，所以更有利于搜索引擎爬虫获取信息，提高搜索结果展现排名。同时，更快的页面加载时间也有利于搜索结果展现排名的提升。
- 实现更加灵活。服务端渲染只是直出页面的初始内容，浏览器端仍然需要做后续工作，以完成页面的最终展现。这样服务端渲染和浏览器端渲染仍可以平衡，在很大程度上也能实现代码复用。

- 可维护性更强。因为借助 React 等类库，我们完全能够实现大范围的代码复用，避免了服务端和浏览器端同时维护两套代码或逻辑。因此，整体代码量更少，维护成本更低。
- 对于低端机型更加友好。因为内容的初步渲染是在服务端完成的，所以对于低端机型更加友好，不至于页面加载时出现白屏幕的状况。
- 对于恶劣的网络环境更加友好。传统的前后端分离方式，在所有的 JavaScript 脚本下载并执行完毕后，才会呈现页面内容，中间经历了较多的网络请求，在恶劣的网络环境下，无疑增加了页面呈现基本内容的难度。在这方面，同构应用显然更有优势。
- 更好的用户体验。为了更加合理地平衡服务端和浏览器端渲染内容，我们可以将页面重要的核心部分设计在服务端完成，而次重要的交互部分可以在更重要的内容渲染完毕后，由浏览器端渲染或实现，这将有力地提升用户体验。

同构的劣势如下：

- 服务端处理的逻辑增多，增加了复杂性。
- 服务端无法完全复用浏览器端代码。
- 增加了服务端的 TTFB（Time To First Byte）时间。TTFB 时间指的是从浏览器发起最初的网络请求，到从服务器接收到第一个字节的这段时间。它包含了 TCP 连接时间、发送 HTTP 请求的时间和获得响应消息的第一个字节的时间。因为对数据的获取和对页面初始内容的渲染，势必会降低服务端返回的速度。

5.3 使用 React 和 Redux 实现同构应用

为了实现服务端渲染，打造同构应用，React 也实现了相应的 API。我们可以在 `react-dom/server` 中找到线索：

```
import ReactDOMServer from 'react-dom/server';
```

依靠 React 提供的 ReactDOMServer 对象可以实现服务端渲染。ReactDOMServer 对象主要提供了 `renderToString()` 和 `renderToStaticMarkup()` 方法。

注意：这两个方法只能在服务端使用。

5.3.1 使用 renderToString 和 renderToStaticMarkup 方法

我们先来看看常用的 renderToString 方法的使用：

```
ReactDOMServer.renderToString(element)
```

该方法接收一个 React element，并将此 element 转化为 HTML 字符串，通过浏览器端返回。因此，在服务端将页面拼接字符串插入 HTML 文档中并返回给浏览器，完成初步服务端渲染的目的。

注意：renderToString 生成的 HTML 字符串的每个 DOM 节点都有一个 data-react-id 属性，根节点会有一个 data-checksum 属性。

这样的属性有什么用呢？事实上，data-react-checksum 属性的值是通过 Adler-32 校验算法实现的。因此，如果两个组件有相同的 props 和 DOM 结构，那么 data-react-checksum 属性的值是一样的。我们知道，服务端渲染完页面内容之后，浏览器端也会渲染组件，以完成组件的交互等逻辑。这时候浏览器端会首先计算出组件的 data-react-checksum 属性的值，如果发现和服务端渲染组件的 data-react-checksum 属性的值相等，则不再进行单独渲染。换句话说，当服务端和浏览器端渲染的组件具有相同的 props 和 DOM 结构时，该 React 组件只会渲染一次。

我们再来看看 renderToStaticMarkup 方法，它和 renderToString 方法又有什么区别呢？使用 renderToStaticMarkup 方法渲染的组件不会带有 data-react-checksum 属性。因此在这种情况下，浏览器端必定会重新渲染组件，最终覆盖服务端的初始组件。浏览器端初始化组件流程图如图 5-5 所示。

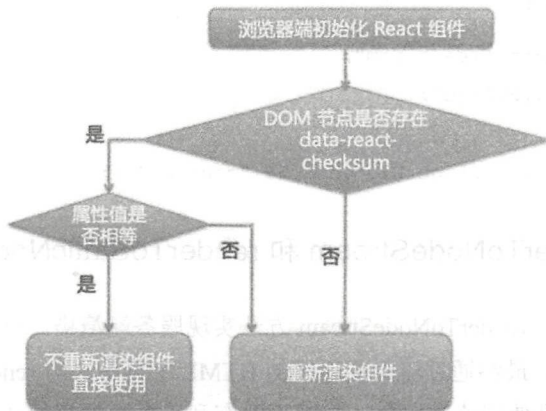


图5-5 浏览器端初始化组件流程图

以上关于 `data-react-*` 属性的介绍，是基于 React 16 版本以前的内容的。值得注意的是，在 React 16 中通过 `renderToString` 渲染的组件不再有 `data-react-*` 属性。因此浏览器端的渲染方法也无法简单地通过 `data-react-checksum` 来判断是否需要重复渲染。

为此，ReactDOM 提供了一个新的 API: `ReactDOM.hydrate()`，它的用法和 `render()` 并无差异，但是可以和 `renderToString` 相互配合使用。在这样的情况下，浏览器端在渲染组件时，将会最大限度地保留在服务端使用 `renderToString()` 所生成的内容结构，并添加浏览器端所特有的事件绑定和交互等。

当然，React 16 是向下兼容的，浏览器端在重新渲染服务端组件时，使用 `render()` 仍然没有问题。但不论是面向未来，还是基于性能的考虑，都应该采用更好的模式。

服务端代码如下：

```
import { renderToString } from "react-dom/server";
import MyPage from "../MyPage";

app.get("/", (req, res) => {
  res.write("<!DOCTYPE html><html><head><title>My Page</title></head><body>");
  res.write("<div id='content'>");
  res.write(renderToString(<MyPage/>));
  res.write("</div></body></html>");
  res.end();
});
```

浏览器端代码如下：

```
import { hydrate } from "react-dom";
import MyPage from "../MyPage";

hydrate(<MyPage/>, document.getElementById("content"));
```

5.3.2 使用 `renderToNodeStream` 和 `renderToStaticNodeStream` 方法

React 16 还提供了 `renderToNodeStream` 方法实现服务端渲染。该方法将持续产生字节流，返回 `Readable stream`。最终通过流形式返回的 HTML 字符串，与 `renderToString` 返回的 HTML 字符串并无差别。但是使用流的概念，显然更加有利于页面的初始加载速度和首屏展现时间。也就是说，服务端处理内容时是实时向浏览器端传输数据，而不是一次性处理完成后才开始向

浏览器端返回结果的。这样做的好处是可以缩短 TTFB 时间。

`renderToStaticNodeStream` 方法和 `renderToNodeStream` 类似，但是仍然不会产生 `data-react-*` 属性。因此，对于典型的静态内容输出页面，采用 `renderToStaticNodeStream` 方法是一个不错的选择。当然，当页面组件需要较多的交互逻辑时，`renderToNodeStream` 显然是更好的选择。

了解了上述方法后，在实际开发中还有一些事项需要考虑。

- 因为在服务端并不存在支持组件挂载的浏览器环境，所以 React 组件只有 `componentDidMount` 之前的生命周期方法有效。因此在 `getInitialState`、`render` 等组件方法中不能用到浏览器的一些特性，比如访问 `localStorage`、`window` 等。合理的做法是，将依赖浏览器环境的操作放到 `componentDidMount` 中处理。
- 在服务端拉取数据后，在很多场景下浏览器端也需要拉取数据，进行二次渲染。为了实现更好的代码复用，一种典型的做法是把请求数据的逻辑放到 React 组件的静态方法中。这样不管是浏览器端还是服务端，在需要获取最新数据时都可以直接访问该方法，以实现代码复用。
- 关于数据请求逻辑的问题：因为服务端不存在 AJAX 的概念，在 Node.js 环境下，一般使用 `http.request` 来完成请求。为了达到代码复用的效果，可以使用 `isomorphic-fetch` 包对请求逻辑的服务端和浏览器端的一致性进行封装。
- 在不能实现代码复用的情况下，可以先进行服务端和浏览器端的判断，为浏览器端和服务端分别准备两套逻辑。比如，根据“`window` 是浏览器端所特有的对象”这一特点，进行环境和逻辑区分。代码如下：

```
var onServer = typeof window === 'undefined';
if (onServer) { // 服务端逻辑 } else { // 浏览器端逻辑 }
```

除此之外，还可以借助于 Webpack 配置来完成前后端的判断，这里不再赘述。

5.3.3 Redux 搭配 React 实现服务端渲染

很多基于 React 的应用都使用 Redux 进行状态管理，我们在前面章节中也重点介绍了 Redux 的内容。同时我们了解到在服务端渲染的情况下，很多场景都要求数据请求在服务端完成。由此可知，Redux 在服务端的使用也屡见不鲜。在这种情况下，当服务端完成预加载数据后，便也实现了 store 的创建，希望浏览器也可以访问并使用 store。

关于在服务端使用 Redux 常见的套路和做法，我们按照两大步骤来介绍。

(1) 在服务端拉取数据，创建 store 实例。

```
// 省略拉取数据的逻辑
```

```
// 创建新的 Redux store 实例
```

```
const store = createStore(App);
```

(2) 将组件渲染成字符串输出。

```
const html = renderToString(react element)
```

```
res.send(`
```

```
<html>
```

```
<body>
```

```
<div id="root">${html}</div>
```

```
<script>
```

```
  window.__INITIAL_STATE__ = ${JSON.stringify(store.getState())}
```

```
</script>
```

```
<script src="/static/bundle.js"></script>
```

```
</body>
```

```
</html>
```

```
`);
```

在这一步中，格外需要注意的一处细节为：

```
window.__INITIAL_STATE__ = ${JSON.stringify(store.getState())};
```

因为 store 是在服务端初始创建的，为了保持两端中的 store 一致，上面代码将 store 的初始值添加到 window 对象的 INITIAL_STATE 属性中。

值得一提的是，此处将 initialState 输出到页面中，从安全角度来看这是十分危险的，需要注意防范 XSS 攻击，一般常用的做法是使用 serialize-javascript 包来处理。

因此，Redux 在服务端唯一要做的事情就是创建 store，并提供应用所需的初始状态。在浏览器端需要配合的工作就是使用服务端返回的状态（window.__INITIAL_STATE__）创建并初始化一个全新的 Redux store。

```
// 通过服务端注入的全局变量得到初始状态
```

```
const initialState = window.__INITIAL_STATE__;
```

```
// 使用初始状态创建 Redux store
```

```
const store = createStore(App, initialState);
```

```
render(
```

```
<Provider store={store}>
```

```
<App />
```

```
</Provider>,  
document.getElementById('root')  
)
```

至此，就介绍完了同构应用开发所需的理论基础。纸上得来终觉浅，后面我们将通过一个简单的例子来加深了解。

5.4 React 16 在服务端渲染上的惊喜

前面章节已经介绍过 React 16 版本在服务端渲染上的一些变动，本节将对此进行简要总结。

- 在浏览器端渲染组件需要配合服务端使用 `hydrate` 方法以区分 `render` 方法。
- React 16 在服务端渲染中提供了支持 `stream` 方式的接口。
- 与浏览器端的新特性类似，服务端渲染除能处理 `React element` 外，也可以处理字符串、数组、数字类型。我们可以将字符串、数组、数字作为参数传递给 `renderToString` 方法。

```
res.write(renderToString([  
  first element  
,  
  second element  
])); res.write(renderToString("hey there")); res.write(renderToString(2));
```

当然，`render` 方法的这个新特性对浏览器端也同样适用。

- React 16 在服务端生成 HTML 字符串时更加高效。比如在它返回的结果 DOM 中废除了 `data-react-checksum` 等属性。
- React 16 允许在渲染 DOM 时加入非标准的 DOM 属性。
- React 16 在服务端渲染并不支持错误处理和 `Portals`（插槽）新特性。
- React 16 在服务端渲染更加迅速。

如图 5-6 所示，我们发现不论在哪个 Node.js 版本下，React 16 在服务端渲染的速度优势都很明显。事实上，React 在服务端渲染、构建同构应用的表现将会更加完美。同构应用在 React 生态中扮演着越来越重要的角色。

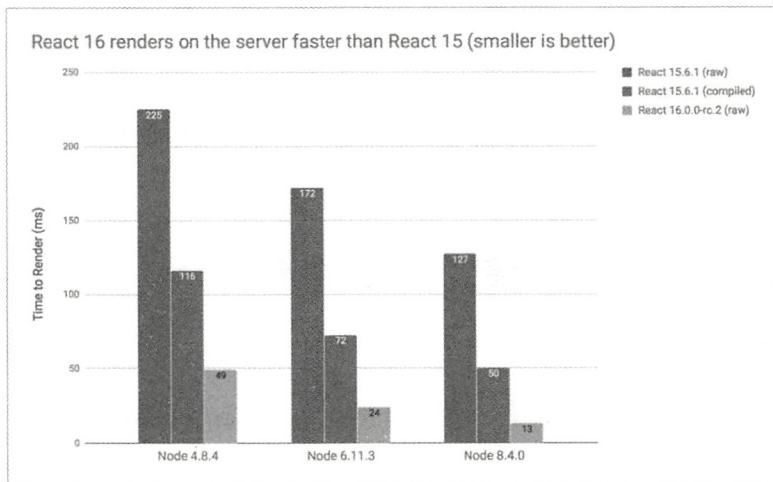


图5-6 React 16和React 15在服务端渲染性能对比

5.5 同构项目实战：基于 Node.js 的“渐进式”流渲染

本书我们将通过一个简单的前后端同构项目，串联起前面章节的内容。在这个项目中，我们将演示 ReactDOMServer 对象的各种方法的使用。为了降低复杂度，暂时不加入 Redux 和前端路由的内容。

5.5.1 项目背景和技术栈介绍

我们使用 React 16 版本，同时采用 Node.js 框架 Express 4.15.3 进行服务端处理。Express 是基于 Node.js 平台的快速、开放、极简的 Web 开发框架，它的用法非常简单。同时使用业界最流行的 Webpack 作为构建工具，基于 Webpack 完成解析、编译、使用插件等工作。

5.5.2 项目目录

如图 5-7 所示，我们将业务代码集中存放在 src 文件夹下。同构项目涉及浏览器端、服务端两大环节，以及两端的同步代码。因此，在 src 文件夹下又分为如下三个文件夹。

- browser: 负责浏览器端的渲染。
- server: 负责服务端的逻辑，主要包括响应页面请求、服务端渲染等。

- **shared:** 此文件夹属于两端同构部分，其中包括构成页面的所有组件、组件依赖的样式文件以及静态资源等。

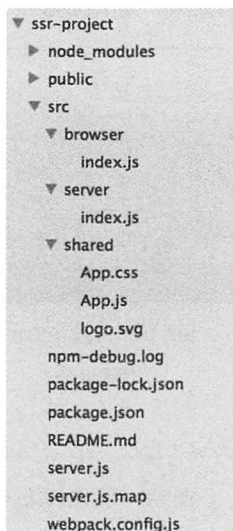


图 5-7 项目目录结构图

根目录下的 `public` 文件夹为 Webpack 的打包目录，对应的 Webpack 配置信息为：

```
entry: "./src/browser/index.js",
output: {
  path: __dirname,
  filename: "./public/bundle.js"
},
```

服务端打包信息为：

```
entry: "./src/server/index.js",
target: "node",
output: {
  path: __dirname,
  filename: "server.js",
  libraryTarget: "commonjs2"
},
```

由此不难发现，在 `webpack.config.js` 配置文件中，我们分别维护了服务端和浏览器端配置逻辑。

5.5.3 代码实现

首先，让我们看一下最终的页面效果，如图 5-8 所示。



图5-8 页面效果图

通过观察可知，页面相对简单，只用一个 App 组件就可以实现。这个组件将会在服务端进行渲染，将结果返回给浏览器后，将会在浏览器端再次进行渲染，并添加浏览器端相应的事件交互：当用户点击按钮之后，会弹出内容为“click event triggered!”的提示信息。

在同构项目中，App 组件将被服务端和浏览器端共享。为此，打开 src/shared/App.js 文件，编写代码如下：

```
import React, { Component } from "react";
import logo from "./logo.svg";
import "./App.css";

class App extends Component {
  constructor(props) {
    super(props);
    this.handleClick = this.handleClick.bind(this);
  }
  handleClick() {
    alert('click event triggered!')
  }
  render() {
```



```

return (
  <div className="App">
    <div className="App-header">
      <img src={logo} className="App-logo" alt="logo" />
      <h2>Welcome to React in the Server</h2>
    </div>
    <p className="App-intro">Isn't this cool? Yes, it's</p>
    <button onClick={e => this.handleClick()}> 请点击按钮 </button>
  </div>
);
}
}

```

```
export default App;
```

完成页面组件的编写后，我们将重点放在服务端逻辑部分。当第一次请求页面时，我们看看服务端具体做了哪些事情。打开 `server/index.js` 文件，其内容如下：

```

import express from "express";
import React from "react";
import { renderToString } from "react-dom/server";
import App from "../shared/App";

const app = express();

app.use(express.static("public"));

app.get("*", (req, res) => {
  const htmlMarkup = renderToString(<App />);
  res.send(`
    <!DOCTYPE html>
    <head>
      <title>Universal React</title>
      <link rel="stylesheet" href="/css/main.css">
      <script src="/bundle.js" defer></script>
    </head>

    <body>

```

```

    <div id="root">${htmlMarkup}</div>
  </body>
</html>
`);
});

app.listen(process.env.PORT || 3000, () => {
  console.log("Server is listening");
});

```

我们对上述代码进行分析。首先生成 `express` 实例，赋值给变量 `app`。

```
const app = express();
```

然后将静态文件目录设置为：项目根目录+`public`。

```
app.use(express.static("public"));
```

因此，对于两个静态文件，可以准确地找到资源位置。

```

<link rel="stylesheet" href="/css/main.css">
<script src="/bundle.js" defer></script>

```

`app.get` 是基本的路由方法，如下代码表示对该域下所有的访问链接做出一致的响应逻辑。

```

app.get('*', function (req, res) {
  ...
});

```

其中，第二个参数是一个回调函数，该函数的第一个参数 `req` 表示从浏览器端发来的 HTTP 请求，第二个参数 `res` 代表返回给浏览器端的响应，这两个参数都是 JavaScript 对象。

在回调函数内部，使用 `res.send` 方法向浏览器端发送一个字符串。

```
res.send(...);
```

这个字符串通过 `renderToString` 方法生成。

```
const htmlMarkup = renderToString(<App/>);
```

最后调用 `app` 实例方法 `listen`，让其监听事先设定的端口（3000）。

```

app.listen(process.env.PORT || 3000, () => {
  console.log("Server is listening");
});

```

至此，就完成了基本的同构应用中服务端渲染部分。

请别忘了，在浏览器端还需要完成页面按钮的点击交互操作。打开 `./src/browser/index.js` 文件，其内容如下：

```
import React from "react";
import { hydrate } from "react-dom";
import App from "../shared/App";

hydrate(<App />, document.getElementById("root"));
```

其中，id 为 root 的 DOM 节点就来自服务端返回的结果。我们也应用了 React 16 的 `hydrate` 方法来完成浏览器端的逻辑部分。

当在浏览器设置中选中“Disable JavaScript（禁用 JavaScript）”选项时（见图 5-9），依然能够得到首屏内容，它是完全由服务端返回的。

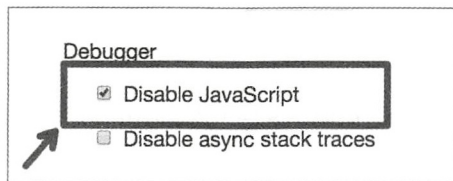


图5-9 选中禁用JavaScript脚本选项

如果把如下代码：

```
import { hydrate } from "react-dom";
hydrate(<App />, document.getElementById("root"))
```

换成

```
import { render } from "react-dom";
render(<App />, document.getElementById("root"));
```

React 也实现了向下兼容，页面一切正常，但是会给出如图 5-10 所示的警告信息。

```
Warning: render(): Calling ReactDOM.render() to hydrate server-rendered markup will stop working in React v17. Replace the
ReactDOM.render() call with ReactDOM.hydrate() if you want React to attach to the server HTML.
printWarning @react-dom.development.js:13921
lowPriorityWarning$1 @react-dom.development.js:13940
renderSubtreeIntoContainer @react-dom.development.js:17098
render @react-dom.development.js:17129
(anonymous) @index.js:6
__webpack_require__ @bootstrap.874b268...:19
module.exports @bootstrap.874b268...:62
(anonymous) @bundle.js:66
```

图5-10 警告信息

为了验证服务端渲染的正确性，我们暂时将 `browser/index.js` 文件中的内容全部注释掉，页

面信息全部由服务端提供。实际上，也得到了如图 5-10 所示的页面信息。不过不出所料，这时候点击页面上的按钮，并不会出现预期的交互效果。

另外，需要说明的是，采用 React 16 版本，在服务端渲染出的页面内容中 DOM 节点不包含 data-react-* 属性，这一切都符合预期（见图 5-11）。



图5-11 页面结构图

最后，我们再演示一下 React 16 的 renderToNodeStream 方法的使用。修改 server/index.js 文件内容如下：

```

import express from "express";
import React from "react";
import { renderToNodeStream } from "react-dom/server";
import App from "../shared/App";

const app = express();

app.use(express.static("public"));

app.get("/*", (req, res) => {
  res.write(`
    <!DOCTYPE html>
    <head>
      <title>Universal React</title>
      <link rel="stylesheet" href="/css/main.css">
      <script src="/bundle.js" defer></script>
    </head>`
  );
  res.write("<div id='root'>");

```

```

const stream = renderToNodeStream(<App/>);
stream.pipe(res, { end: false });
stream.on('end', () => {
  res.write("</div></body></html>");
  res.end();
});
});

app.listen(process.env.PORT || 3000, () => {
  console.log("Server is listening");
});

```

注意：上述代码体现了 Node.js 中流的概念。流的精髓在 `pipe()` 方法中有很好的体现，其提供了桥接能力，对数据流的两端（上游/下游或称为读/写流）进行桥接。

返回流的基础在于 React 的 `renderToNodeStream` 方法返回了一个 `Readable stream`，因此完成了渐进式渲染。为了配合返回一个流，使用 `res.write` 方法代替先前的 `res.send` 方法来发送响应内容。渐进式渲染是一个相对较新的概念，它对于首屏信息的输出、用户体验的提升具有重要意义。

对比常用的 `renderToString` 方法，在使用 `renderToNodeStream` 的场景下，页面的 TTFB 时间可以明显缩短。这是因为 TTFB 时间是服务器响应首字节的时间，采用流的渐进式渲染可以最大限度地缩短服务器响应时间，从而使浏览器可以更快地接收到信息。

对比结果，在使用 `renderToString` 完成渐进式渲染的基础上，TTFB 时间为 12.07ms，如图 5-12 所示。

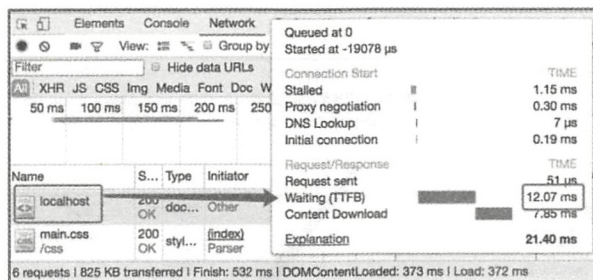


图5-12 渐进式渲染得到的TTFB时间

在使用 `renderToString` 方法进行传统方式渲染的情况下，TTFB 时间为 53.44 ms，如图 5-13 所示。

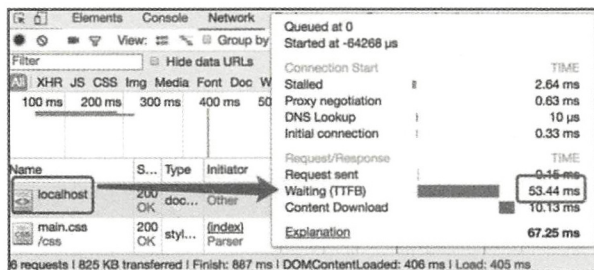


图5-13 传统方式渲染得到的TTFB时间

综上所述，通过渐进式渲染，用户较快地看到了页面内容。

5.5.4 同构应用与浏览器端渲染优势对比

本节我们再来介绍一下在同构方式下首屏时间、页面加载时间相比于浏览器端渲染的优势。我们选取 DOMContentLoaded 时间以及页面 Load 时间作为例子进行具体说明。

当初始的 HTML 文档被完全加载和解析完成之后，DOMContentLoaded 事件被触发，而无须等待样式表、图片和子框架加载完成；Load 事件用于检测页面是否完全加载完成。

具体来说，如果 HTML 文档中包含脚本，则脚本会阻塞文档的解析，在处理完脚本之后，浏览器再继续解析 HTML 文档。在任何情况下，触发 DOMContentLoaded 事件都不需要等待图片等其他资源加载完成。

由此可知，使用服务端渲染的同构应用，在 DOMContentLoaded 和 Load 这两个时间指标上都应该体现出性能优势。当使用 renderToString 方法进行服务端渲染时，得到的时间指标数据如图 5-14 所示。

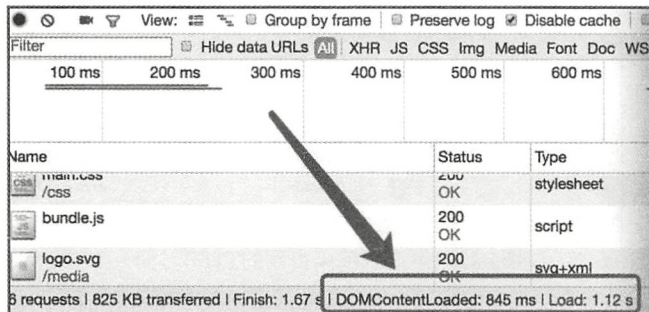


图5-14 时间指标数据（1）

接下来，我们将 `server/index.js` 中的服务端渲染内容注释掉，只保留最初的根节点。

```
import express from "express";
// import React from "react";
// import { renderToString } from "react-dom/server";
// import App from "../shared/App";
```

```
const app = express();
```

```
app.use(express.static("public"));
```

```
app.get("*", (req, res) => {
```

```
  // const htmlMarkup = renderToString(<App />);
```

```
  res.send(`
```

```
    <!DOCTYPE html>
```

```
    <head>
```

```
      <title>Universal React</title>
```

```
      <link rel="stylesheet" href="/css/main.css">
```

```
      <script src="/bundle.js" defer></script>
```

```
    </head>
```

```
    <body>
```

```
      <div id="root"></div>
```

```
    </body>
```

```
  </html>
```

```
`);
```

```
});
```

```
app.listen(process.env.PORT || 3000, () => {
```

```
  console.log("Server is listening");
```

```
});
```

得到的时间指标数据如图 5-15 所示。

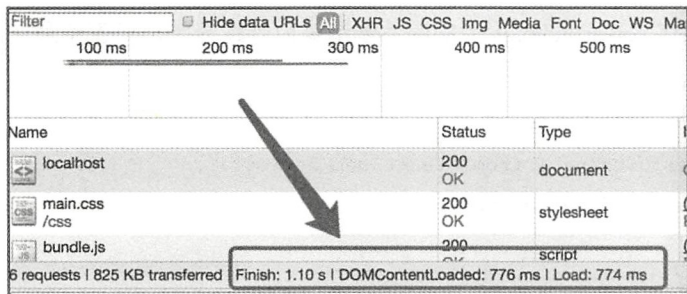


图5-15 时间指标数据（2）

可见，在使用服务端渲染的情况下，DOMContentLoaded 时间由 845ms 缩短为 776ms，提升了 8%以上；页面 Load 时间由 1.12s 缩短为 774ms，提升了 31%左右。

5.6 Next.js 设计理念和使用

Next.js 是一个基于 React 服务端渲染构建同构应用的框架，其精妙的理念设计和灵活的功能集合，一经推出便受到热捧。本节我们将会了解 Next.js 的使用和设计思想。

5.6.1 Next.js 的极简理念

当开发者决定使用 React 进行项目开发时，就得先使用 Webpack 配置一系列必要功能，构建工具的学习曲线急剧提升。对于同构应用，服务端渲染必备的环境以及基本知识，也往往令初学者望而生叹。

而 Next.js 基于 React + Webpack + Babel，已经为开发者做好了一切环境准备。下面我们来看看如何方便快捷地开启 Next.js。首先安装依赖：

```
$ npm install next --save
```

进入 pages 文件夹：

```
$ mkdir pages
```

建立 pages/index.js 文件：

```
import React from 'react'
export default () => <div>Hello world!</div>
```

在 package.json 中，添加如下脚本命令：

```
{  
  "scripts": {  
    "dev": "next"  
  }  
}
```

最后运行：

```
$ npm run dev
```

在安装完 Next.js 之后，开发者完全不需要再进行配置，更不需要安装其他依赖，一切变得再简单不过了。

5.6.2 Next.js 设计思想

很多年以来，Next.js 开发团队一直都在研究同构应用这个话题。随着 Node.js 的发展，服务端和浏览器端共用代码已经具备了基本土壤。为此，前端开发者进行了很多尝试，比如选择 Express 作为 Node.js 框架、Jade 作为模板引擎。如此，服务端渲染了基本 HTML，直出给浏览器，之后浏览器端便接管一切，只需要加入特定的页面交互即可实现最终效果。

但这样的做法与传统做法并没有太大区别，前端和后端在分工本质上仍然要进行拆分。而 React 的出现打破了这一藩篱，它提供了纯 render 函数，根据数据表达 UI，不同于传统的模板引擎，React 创新的关键之一在于组件生命周期，组件生命周期函数允许我们在传统的服务端渲染的基础上，在浏览器端为后续交互做进一步处理。

Next.js 依托此思想，诞生了自己的哲学，体现了以下几大特点。

1. 零配置和使用文件系统作为 API

关于零配置，是指 Next.js 隔离开 Webpack 的配置，已经为开发者做好了一切。当然，它也支持一些自定义设置。另外一大亮点就是使用了文件系统，就像我们开发 Node.js 项目一样，当配置好 package.json 文件之后，所有的依赖都会自动安装到 ./node_modules 目录下。

Next.js 带来的创新体现在 Next.js 项目中，就是存在一个 pages 目录，我们可以把页面级别的组件放置在该目录下，pages 目录下的组件将会自动映射为一个基于服务器的路由。

例如，pages/index.js 文件自动匹配到网站/路由：

```
import React from 'react'  
export default () => <marquee>Hello world</marquee>
```

pages/about.js 自动匹配到“关于”页面：

```
import React from 'react'
export default () => <h1>About us</h1>
```

这样的自动配置能够节省开发者的很多时间，即使不具备 Node.js 的基本知识，也可以按照此“约定”进行开发。当然，如果项目需要更加复杂的路由设计，开发者也可以对此“约定”进行拦截和自定义。

2. 关于数据获取

要完成服务端渲染，在服务端进行数据请求和获取必不可少。为此，Next.js 扩展了 `ReactComponent` 的 `getInitialProps` 方法，我们一般使用该方法来获取数据。

```
import React from 'react'
import 'isomorphic-fetch'
export default class extends React.Component {
  static async getInitialProps () {
    const res = await fetch('https://api.company.com/user/123')
    const data = await res.json()
    return { username: data.profile.username }
  }
}
```

通常 `getInitialProps` 方法的返回值是异步获取数据的结果，它将会默认扩展成该组件的 `props` 一部分。

另外，Next.js 对 `getInitialProps` 方法的改变还体现在其参数还接收了一个上下文对象，该对象具有如下属性：

{ req: HTTP 请求对象（服务端渲染独有），res: HTTP 响应对象（服务端渲染独有），pathname: URL 中的路径部分，query: URL 中的查询字符串部分解析出的对象，err: 错误对象，渲染时发生的错误信息，xhr: XMLHttpRequest 对象（浏览器端渲染独有）}

因此，在此方法中，我们可以对服务端和浏览器端环境进行判断，并加以区分。如下根据不同的环境，返回访问 UA 的信息。

```
export default class extends React.Component {
  static async getInitialProps ({ req }) {
    return req
      ? { userAgent: req.headers['user-agent'] }
      : { userAgent: navigator.userAgent }
  }
}
```

```
render () {
  return <div>
    Hello World {this.props.userAgent}
  </div>
}
```

3. Next.js 的路由和代码分割

`pages` 目录下的组件页面都会自动进行服务端渲染。Next.js 提供了 `Link` 组件来实现页面跳转及路由功能。

例如，如果需要从首页 `pages/index` 跳转到 `pages/about` 页面，便可以在首页中添加 `Link` 组件进行页面切换。

```
import Link from 'next/link'
export default () => (
  <div>Click <Link href="/about"><a>here</a></Link> to read more</div>
)
```

类似的，在 `next/prefetch` 模块中还有一个具有 `prefetch` 功能的 `Link` 组件。

```
import Link from 'next/prefetch'
```

Next.js 还提供了 `Router` 对象来满足命令式的写法。

```
import Router from 'next/router'

export default () => (
  <div>Click <span onClick={() => Router.push('/about')}>here</span> to read more</div>
)
```

通过这些功能，我们可以动态地加载页面，即使一个页面引用了大量资源，也不会影响其他页面的加载时间。

4. Next.js 支持 CSS-in-JS

Next.js 使用 `glamor` 包进行样式管理。

```
import React from 'react'
import css from 'next/css'

export default () => <p className={style}>Hi there!</p>
```

```
const style = css({
  color: 'red',
  ':hover': {
    color: 'blue'
  },
  '@media (max-width: 500px)': {
    color: 'rebeccapurple'
  }
})
```

官方认为这样的代码结构能够最大化实现性能优化,也有利于基于流的渐进式服务端渲染。同时,其他的 CSS-in-JS 解决方案也都可以灵活支持。

5.7 使用 Next.js 实现同构应用

本节尝试使用 Next.js 构建一个简单的服务端渲染 React 应用。我们知道,React App 通过虚拟的 DOM 实现了对真实 DOM 的抽象,这样的设计迅速引领了前端开发浪潮。但是虚拟的 DOM 带来了一些弊端,比如在前后端分离的开发模式下,SEO 就成了问题;首屏加载时间变长,各种加载消磨了人的耐心。我们查看页面源代码,页面结构大体如图 5-16 所示。

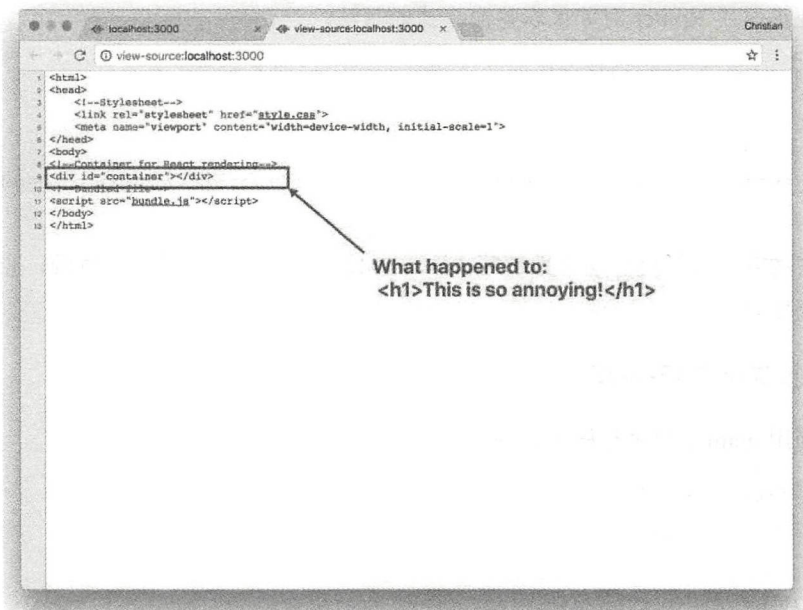


图5-16 页面结构图(浏览器渲染的白屏情况)

初始加载的 HTML 文档并没有任何实际内容，实际内容都是通过 JavaScript 脚本渲染的。

下面将使用 Football Data API 来开发一个简单的基于 Next.js 的应用，这个应用将展现英超联赛的实时积分榜，并包含了简单的路由开发和页面跳转。

相信所有的开发者都不喜欢长时间安装和配置各种依赖、插件。幸运的是，Next.js 作为一个独立的 npm 包，最大限度地帮助完成了很多耗时且无趣的工作。首先需要安装 Next.js。

```
# Start a new project
npm init
# Install Next.js
npm install next@beta react react-dom --save
```

安装结束后，开启脚本。

```
"scripts": {
  "start": "next"
},
```

完整的项目目录结构如图 5-17 所示。

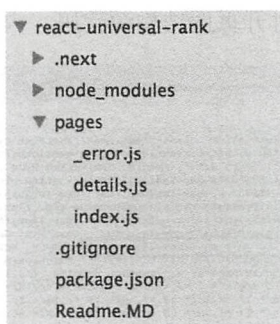


图5-17 完整的项目目录结构图

然后在根目录下创建一个 pages 文件夹，并在这个文件夹下新建一个 index.js 文件。

```
// ./pages/index.js

// 导入 React
import React from 'react'

// 导出一个匿名箭头函数，这个函数会返回如下模板
export default () => (
  <h1>This is just so easy!</h1>
)
```

现在在命令行直接输入以下命令：

```
# Start your app
npm start
```

渲染结果如图 5-18 所示。

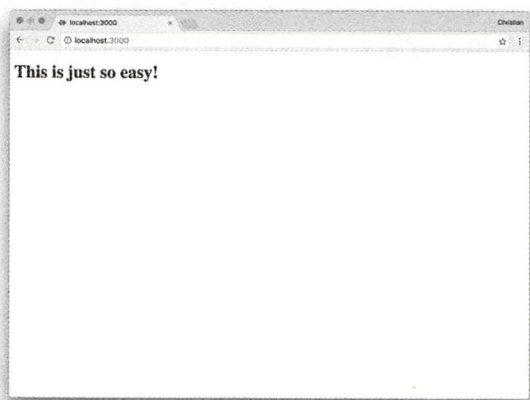


图5-18 渲染结果

验证此页面来自服务端渲染，打开页面结构源代码，内容如图 5-19 所示。

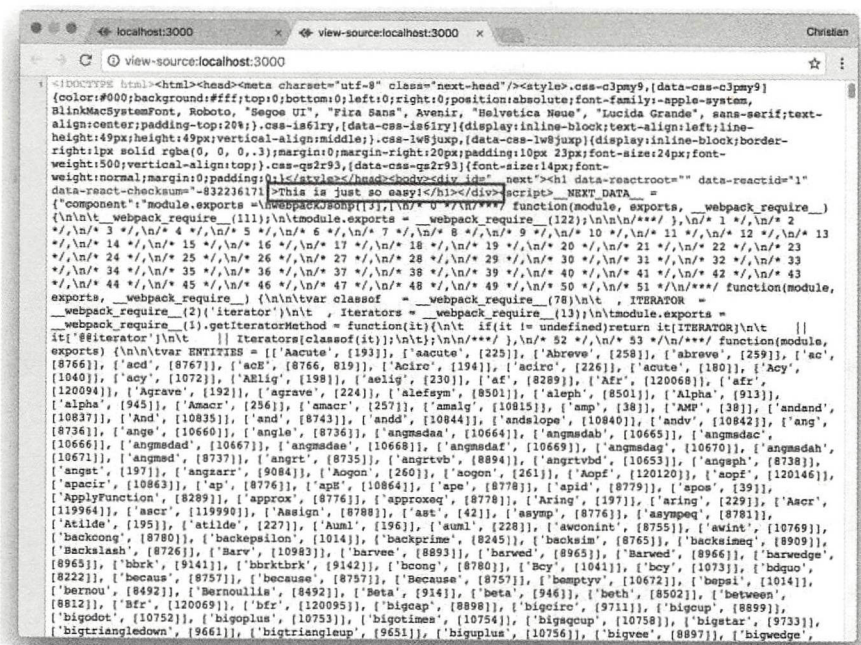


图5-19 服务端渲染内容

我们看到,安装 Next.js 之后,Next.js 会搭建一个基于 React + Webpack + Babel 的构建环境。如果手动实现这一切的话,除了烦琐不说,光 Webpack 配置、node_modules 依赖、Babel 插件等就够折腾半天的了。就这么简单,我们就已经实现了最简单的服务端渲染同构例子。

接下来,在./pages/index.js 文件内,我们可以添加页面 head 标签、meta 信息、样式资源等。

```
// ./pages/index.js
import React from 'react'
// 导入 Head 组件
import Head from 'next/head'

export default () => (
  <div>
    <Head>
      <title>League Table</title>
      <meta name="viewport" content="initial-scale=1.0, width=device-width" />
      <link rel="stylesheet" href="https://unpkg.com/purecss@0.6.1/build/pure-min.css" />
    </Head>
    <h1>This is just so easy!</h1>
  </div>
)
```

这个 Head 当然不是指真实的 DOM,千万别忘了 React 虚拟的 DOM 的概念。其实这是 Next.js 提供的 Head 组件,不过最终一定会被渲染成真实的 head 标签的。

该应用将会请求英超联赛积分榜的实时信息,对于异步请求,Next.js 也提供了特殊方法来完成两端同构。Next.js 提供了组件生命周期钩子方法 getInitialProps(),使得框架能够在服务器上进行初始渲染,如果需要的话,还可以在客户端继续进行渲染。这个方法支持异步选项,并且是服务端/客户端同构的。我们可以使用 async/await 方式,处理异步请求。请看下面的示例。

```
import React from 'react'
import Head from 'next/head'
import axios from 'axios';

export default class extends React.Component {
  // Async operation with getInitialProps
  static async getInitialProps () {
```

```

    // res is assigned the response once the axios
    // async get is completed
    const res = await axios.get('http://api.football-data.org/v1/competitions/426/
leagueTable');
    // Return properties
    return {data: res.data}
  }
}

```

在 `getInitialProps` 方法内，使用了 `axios` 类库来发送 HTTP 请求。网络请求是异步的，因此需要在未来某个合适的时候（当请求结果返回时）接收数据。这里使用先进的 `async/await`，以同步的方式处理，从而避免了回调嵌套和 `Promise` 链。

将异步获得的数据返回，它将自动挂载在 `props` 上，在 `render` 方法中便可以通过 `this.props.data` 获取数据。

```

import React from 'react'
import Head from 'next/head'
import axios from 'axios';

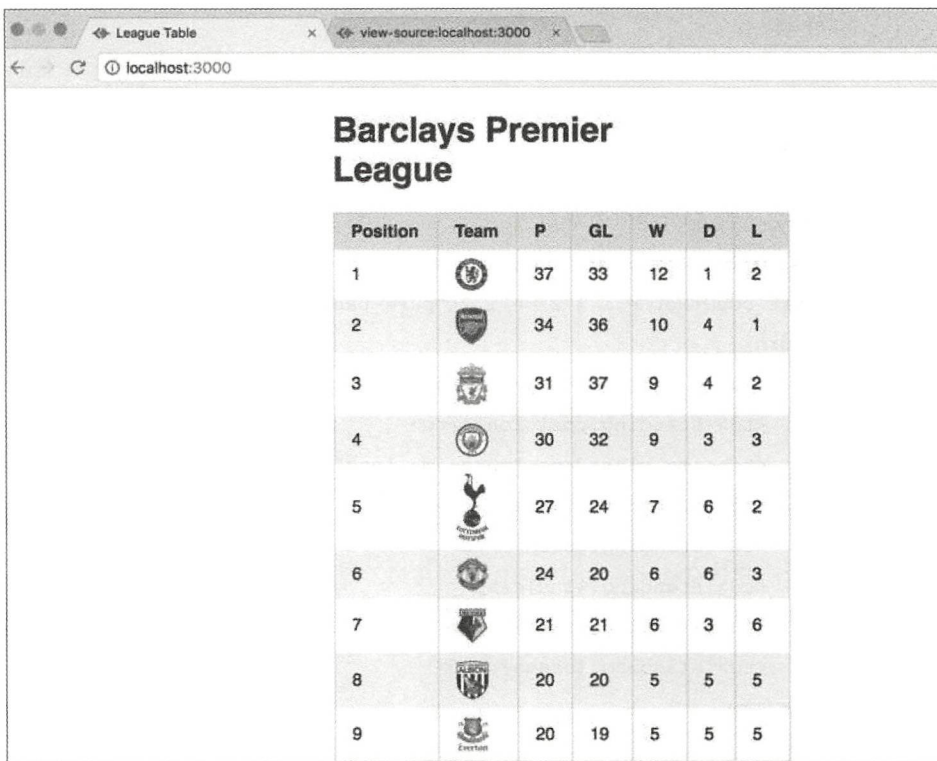
export default class extends React.Component {
  static async getInitialProps () {
    const res = await axios.get('http://api.football-data.org/v1/competitions/426/
leagueTable');
    return {data: res.data}
  }
  render () {
    return (
      <div>
        <Head>
          .....
        </Head>
        <div className="pure-g">
          <div className="pure-u-1-3"></div>
          <div className="pure-u-1-3">
            <h1>Barclays Premier League</h1>
            <table className="pure-table">

```

```
<thead>
  <tr>
    .....
  </tr>
</thead>
<tbody>
{this.props.data.standing.map((standing, i) => {
  const oddOrNot = i % 2 == 1 ? "pure-table-odd" : "";
  return (
    <tr key={i} className={oddOrNot}>
      <td>{standing.position}</td>
      <td><img className="pure-img logo" src={standing.crestURI}/></td>
      <td>{standing.points}</td>
      <td>{standing.goals}</td>
      <td>{standing.wins}</td>
      <td>{standing.draws}</td>
      <td>{standing.losses}</td>
    </tr>
  );
}})
</tbody>
</table>
</div>
<div className="pure-u-1-3"></div>
</div>
</div>
);
}
```

再次访问页面，效果如图 5-20 所示。

最后，我们将实现应用页面的路由和跳转。Next.js 不需要任何额外的路由配置信息，只需要在 `pages` 文件夹下新建文件即可，每一个文件都将是一个独立的页面。`pages` 文件夹中的每一个组件都将会自动映射为一个基于服务器的路由。比如 `pages/detail.js` 组件将会自动映射成 `/detail` 这个 URL。












Position	Team	P	GL	W	D	L
1		37	33	12	1	2
2		34	36	10	4	1
3		31	37	9	4	2
4		30	32	9	3	3
5		27	24	7	6	2
6		24	20	6	6	3
7		21	21	6	3	6
8		20	20	5	5	5
9		20	19	5	5	5

图5-20 页面效果

现在我们来新建一个球队详情页面。新建./pages/detail.js 文件如下：

```
// ./pages/detail.js
import React from 'react'
export default () => (
  <p>Coming soon. . .!</p>
)
```

使用 Next.js 已经准备好的组件来现实页面跳转。

```
// ./pages/detail.js
import React from 'react'

// Import Link from next
import Link from 'next/link'

export default () => (
```



```
<div>
  <p>Coming soon. . .!</p>
  <Link href="/"><a>Go Home</a></Link>
</div>
)
```

这个页面不能总是显示“Coming soon...!”信息，我们来进行完善以展示更多的内容。通过页面 URL 的参数 id 变量，来实现并展现当前相应球队的信息。

```
import React from 'react'
import Head from 'next/head'
import Link from 'next/link'
import axios from 'axios';

export default class extends React.Component {
  static async getInitialProps ({query}) {
    // Get id from query
    const id = query.id;
    if(!process.browser) {
      // Still on the server so make a request
      const res = await axios.get('http://api.football-data.org/v1/competitions/426/leagueTable')
      return {
        data: res.data,
        // Filter and return data based on query
        standing: res.data.standing.filter(s => s.position == id)
      }
    } else {
      // Not on the server just navigating so use
      // the cache
      const bplData = JSON.parse(sessionStorage.getItem('bpl'));
      // Filter and return data based on query
      return {standing: bplData.standing.filter(s => s.position == id)}
    }
  }

  componentDidMount () {
    // Cache data in localStorage if
    // not already cached
    if(!sessionStorage.getItem('bpl')) sessionStorage.setItem('bpl', JSON.stringify(
      (this.props.data)
    ))
  }
}
```

```
    }  
  
    // . . . render method truncated  
  }  
}
```

这个页面根据 `query` 变量，动态展示指定球队的信息。具体来看，使用 `getInitialProps` 方法获取 URL 中 `id` 参数值，根据 `id` 筛选出（使用 `filter` 方法）展示信息。因为一支球队的信息比较稳定，所以在客户端使用了 `sessionStorage` 进行存储。

完整的 `render` 方法如下：

```
export default class extends React.Component {  
  // . . . truncated  
  render() {  
  
    const detailStyle = {  
      ul: {  
        marginTop: '100px'  
      }  
    }  
  
    return (  
      <div>  
        <Head>  
          <title>League Table</title>  
          <meta name="viewport" content="initial-scale=1.0, width=device-width" />  
          <link rel="stylesheet" href="https://unpkg.com/purecss@0.6.1/build/pure-min.css" />  
        </Head>  
  
        <div className="pure-g">  
          <div className="pure-u-8-24"></div>  
          <div className="pure-u-4-24">  
            <h2>{this.props.standing[0].teamName}</h2>  
            <img src={this.props.standing[0].crestURI} className="pure-img" />  
            <h3>Points: {this.props.standing[0].points}</h3>  
          </div>  
          <div className="pure-u-12-24">  
            <ul style={detailStyle.ul}>  
              <li><strong>Goals</strong>: {this.props.standing[0].goals}</li>  
            </ul>  
          </div>  
        </div>  
      </div>  
    )  
  }  
}
```



```
    <li><strong>Wins</strong>: {this.props.standing[0].wins}</li>
    <li><strong>Losses</strong>: {this.props.standing[0].losses}</li>
    <li><strong>Draws</strong>: {this.props.standing[0].draws}</li>
    <li><strong>Goals Against</strong>: {this.props.standing[0].
goalsAgainst}</li>
    <li><strong>Goal Difference</strong>: {this.props.standing[0].
goalDifference}</li>
    <li><strong>Played</strong>: {this.props.standing[0].playedGames}
</li>

    </ul>
    <Link href="/">Home</Link>
  </div>
</div>
</div>
)
}
```

如图 5-21 和图 5-22 所示, 根据同一页面不同的 query 值, 分别展示了冠军切尔西和曼联球队的信息。

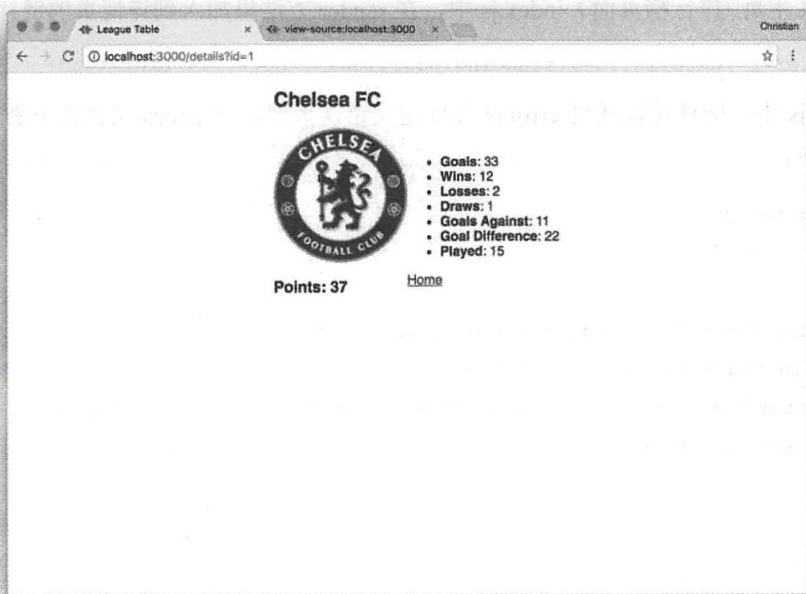


图5-21 页面效果 (1)

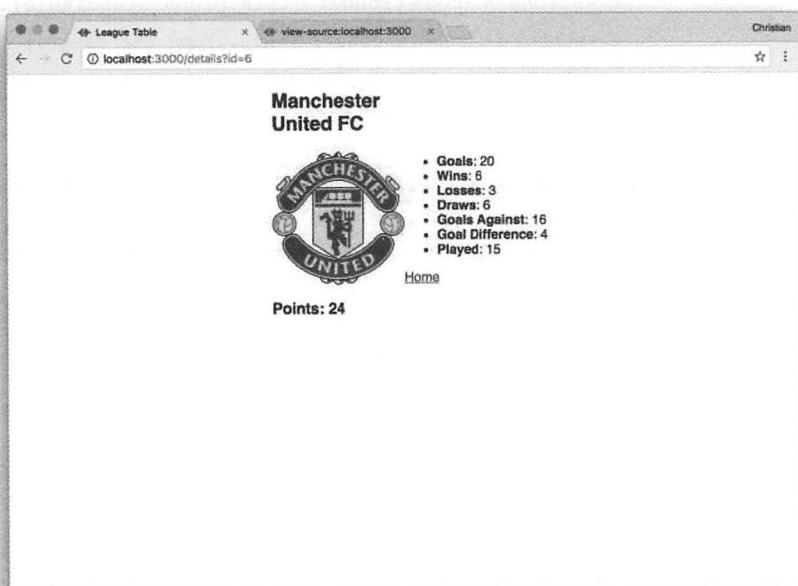


图5-22 页面效果 (2)

别忘了在主页（排行榜页面）index.js 中，在 render 方法里加入到详情页的链接。

```
<td><Link href={` /details?id=${standing.position}`}>More...</Link></td>
```

在 Next.js 中，同样可以通过 error.js 文件定义错误页面。在 pages 文件夹下新建 error.js 文件，内容如下：

```
// ./pages/error.js
import React from 'react'

export default class Error extends React.Component {
  static getInitialProps ({ res, xhr }) {
    const statusCode = res ? res.statusCode : (xhr ? xhr.status : null)
    return { statusCode }
  }

  render () {
    return (
      <p>{
        this.props.statusCode
      }
    )
  }
}
```



```
    ? `An error ${this.props.statusCode} occurred on server`  
    : 'An error occurred on client'  
  }</p>  
)  
}  
}
```

设置 `error.js` 之后，当页面发生 404 错误时，将显示如图 5-23 所示的效果。



图5-23 页面错误效果

除此之外，Next.js 还有很多功能可以使用。比如搭配 `prefetch`，预先请求资源；动态加载组件（Next.js 支持 TC39 动态引入依赖），从而减少首次打包脚本的大小；虽然它替我们封装好了 Webpack、Babel 等工具，但是也可以根据需要自定义。

5.8 本章小结

本章介绍了基于 React 开发同构应用的技术实现。前后端的合作和分工、模式的变迁和不同模式的优缺点，将会是一个永恒的话题。究竟是否需要服务端渲染、是否需要同构应用开发，则应该结合实际场景和需求来选择。

React 16 的设计更加完善和丰富，对同构应用开发的支持更好，这也将是未来发展的一大趋势。在此之上，社区对同构应用也更加关注，二级类库不断涌现。本章介绍的 Next.js 就是一个很好的例子。

React 为同构应用打开了一扇窗户。在 React 同构设计以及 Node.js 迅速发展的背景下，前端开发完全可以拥有更广阔的空间。



第6章

深入理解 React 技术内幕与生态社区

在前面章节中，我们已经从 React、Redux、服务端渲染、同构应用等方面对 React 技术栈进行了讲解。事实上，React 及其相关生态的内容远不止这些，经过几年的进步和持续演进，如今 React 生态可谓“博大精深”。很多开发者能够上手完成项目，但是如何写出更优雅的代码，以及如何体现并应用 React “最佳实践”呢？

在本章中，我们将从以下几个方面进行更深层面的介绍。

- React 设计理念和魔法。
- React 组件的组合和复用。
- React “轮子”开发。
- 简易的 React 库的编写。
- Redux 数据结构优化和角色分析。

结合社区中的优秀思想，以及笔者的实践经验，希望在启发读者的同时，为 React 的进阶开启一扇大门。

6.1 React 组件的组合和复用——高阶组件

复用，在软件工程学中至关重要——巧妙利用复用，以及具有良好的复用设计思路，都可以大幅提高软件开发效率。组合，又是函数式编程推崇的概念——组合意味着把多个函数或者功能原子组合起来，变成一个新的函数或者更加丰富的功能模块。React 的设计思想在很多环节上都体现了函数式编程的思路，同时其倡导的组件化方式，又可以将组合和复用功能发挥到最大程度。本节我们将探讨 React 组件中几种常见的组合和复用方式，这些都是强大的 React 社区



带来的思想精华。

6.1.1 高阶组件注意事项和编写原则

高阶组件并不是 React 带来的特性或者 API，而是基于 React 的设计思想，开发者总结出来的一种优秀组件的编写思路。它能够解决复用问题，组合更多的功能以进行扩展。高阶组件和高阶函数的思想异曲同工，作为 JavaScript 开发者，对高阶函数的概念应该并不陌生。在 JavaScript 语言中，体现高阶函数的场景并不少见，它在真实的开发应用中也屡见不鲜。比如事件节流的设计，在某些场景下，某些事件可能会在短时间内被多次触发。为了保证性能，我们预期事件处理函数及其相关计算逻辑并不是每次都会执行的，常见的做法就是设计一个节流的高阶函数。

```
function throttle(fn, interval) {  
  var doing = false;  
  
  return function() {  
    if (doing) {  
      return;  
    }  
    doing = true;  
    fn.apply(this, arguments);  
    setTimeout(function() {  
      doing = false;  
    }, interval);  
  }  
}  
  
window.onresize = throttle(function(){  
  console.log('resize execute');  
}, 500);
```

`throttle` 函数接收的第一个参数用于定义计算逻辑，第二个参数用于定义最短的重复计算时间。它接收了一个计算函数，并对计算过程进行节流增强，最终返回一个函数。这就是高阶函数的思想体现。在复用层面，我们可以按照如下代码进行操作。

```
window.onscroll = throttle(function(){  
  console.log('scroll execute');  
}, 500);
```

由此可见，高阶组件也是一个函数，它能够接收一个组件，并返回一个新的组件。

```
const NewComponent = higherOrderComponent(WrappedComponent);
```

根据这个特性，我们能够想到这样做的意义在于扩展接收到的 `WrappedComponent`，通过 `higherOrderComponent` 函数对原有组件进行包装，生成并返回一个全新的 `NewComponent` 组件。这个组件组合了原有的 `WrappedComponent` 组件功能，以及新加入的功能或信息。

使用高阶组件需要注意的事项和遵守的原则如下：

- 高阶组件不可以直接修改接收到的组件的自身行为，只能进行功能组合。
- 高阶组件是纯函数，需要保证没有副作用。
- 在进行功能组合时，一般通过增加不相关的 `props` 的形式给原有组件传递信息。
- 不要在 `render` 方法中使用高阶组件。
- 高阶组件不会传递 `refs`。

高阶组件在很多 React 相关类库中被广泛使用。比如 `react-redux` 中的 `connect` 就返回一个高阶组件，读者可以翻阅本书第 4 章进行了解。

6.1.2 高阶组件从场景到应用

本节我们将通过一个真实场景来理解并体会高阶组件的奥秘。该工程代码可以在本书配套的代码仓库中找到。

这里实现了一个通讯录联系人列表，该列表具有搜索过滤功能，如图 6-1 所示。

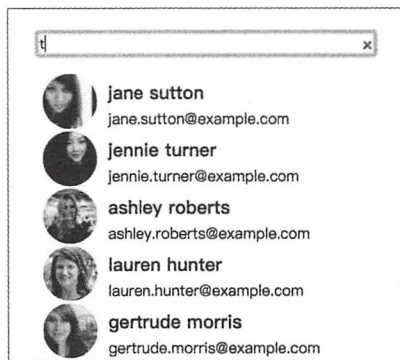


图6-1 通讯录联系人列表

在这样的场景中，联系人列表往往需要通过网络获取。我们先看一下项目目录结构，如图 6-2 所示。

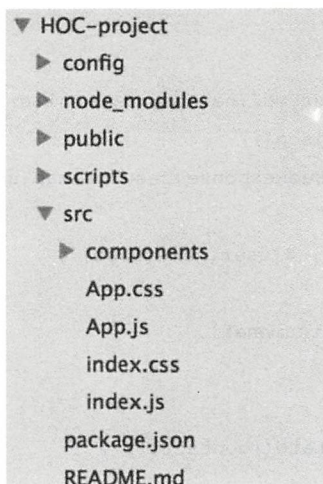


图6-2 项目目录结构图

对应地，我们对页面组件进行拆分，如图 6-3 所示。

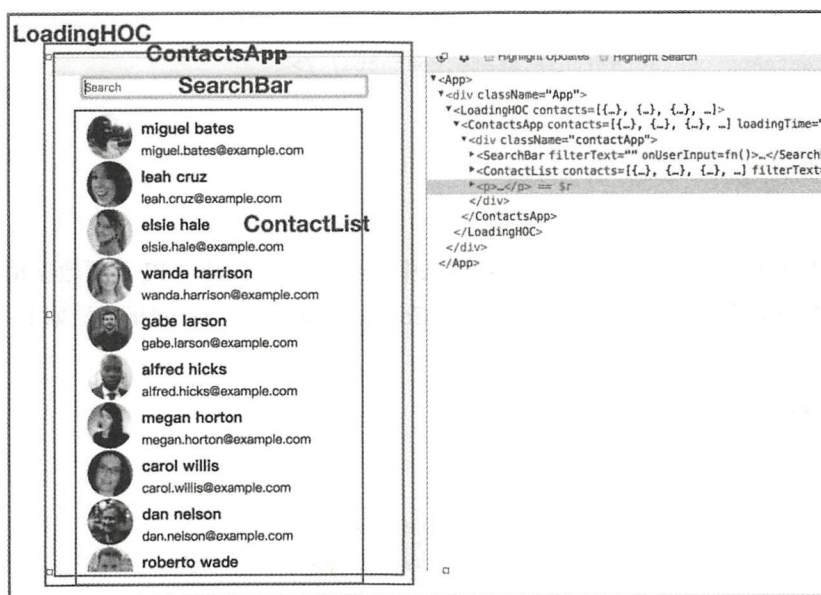


图6-3 组件设计图

`./src/App.js` 文件内容如下：

```
class App extends Component {  
  state = { contacts: [] }  
}
```

```
componentDidMount() {
  fetch('https://api.randomuser.me/?nat=us,gb&results=50')
    .then(response => response.json())
    .then(parsedResponse => parsedResponse.results.map(user => (
      {
        name: `${user.name.first} ${user.name.last}`,
        email: user.email,
        thumbnail: user.picture.thumbnail
      }
    )))
    .then(contacts => this.setState({contacts}));
}

render() {
  return (
    <div className="App">
      <ContactsApp contacts={this.state.contacts} />
    </div>
  );
}
```

在根组件 App 的 `componentDidMount` 方法中，我们通过 `fetch` API 进行数据请求，并将联系人数据作为 `prop` 传递给 `ContactsApp` 组件。`ContactsApp` 组件的 `render` 方法如下：

```
render() {
  const { loadingTime } = this.props;
  return (
    <div className="contactsApp">
      <SearchBar filterText={this.state.filterText}
        onUserInput={this.handleUserInput} />
      <ContactList contacts={this.props.contacts}
        filterText={this.state.filterText}/>
    </div>
  );
}
```

`ContactsApp` 组件渲染搜索框组件 `SearchBar`，以及将 `this.props.contacts` 数据继续向下传递

给 `ContactList` 组件，并最终由 `ContactList` 组件完成数据到 UI 的渲染。

目前这样做并没有什么问题，但是当每次刷新页面后，由于发送请求获取数据可能时间相比较长，会短暂出现没有内容的白页面。为了达到更好的浏览体验，现在添加一个加载图标或提示文本，如图 6-4 所示。

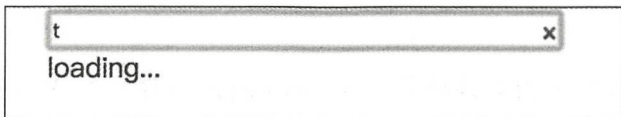


图6-4 加载提示效果

完成这个需求很简单，最直观的想法就是在 `ContactsApp` 组件中，根据 `this.props.contacts` 长度进行选择性渲染。如果其长度为 0，即表示组件还没有接收到数据内容，正处于加载中，此时应当展示加载提示文本，否则正常输出数据内容。

```
render() {  
  const { loadingTime } = this.props;  
  return(  
    <div className="contactsApp">  
      // ...  
      {this.props.contacts.length === 0 ?  
        <div> loading... </div>  
        :  
        <ContactList contacts={this.props.contacts}  
          filterText={this.state.filterText}/>  
      }  
      // ...  
    </div>  
  )  
}
```

如果页面中有很多类似的网络请求都需要添加相同的加载提示文本，那么手动添加将无法达到复用效果。

借助于前文提到的高阶组件的概念，我们试图编写一个加载提示文本的高阶组件。

```
const withLoadingHoc = (WrappedComponent) => {  
  return class extends Component {  
    render() {
```

```

    return this.props.contacts.length === 0 ?
    <div> loading... </div>
    :
    <WrappedComponent {...this.props}/>
  }
}
}

```

最终此高阶组件的使用方式就是将 `ContactsApp` 组件作为 `WrappedComponent` 传入 `withLoadingHoC` 中：

```
withLoadingHoC(ContactsApp);
```

在 `withLoadingHoC` 中，还是使用 `this.props.contacts.length` 进行判断，为此需要将这个限定的 `propname(this.props.contacts)` 进行抽象传递。具体做法是将 `withLoadingHoC` 柯里化，它将会接收两个参数，先后执行。第一个参数是由业务调用方指定的 `propname`，对应于上面场景为 `this.props.contacts`，用于判断数据是否加载完毕。因此，`withLoadingHoC` 在第一次执行后会返回一个函数，并执行原有高阶组件的执行逻辑，即：

```
withLoadingHoC('contacts')(ContactsApp);
```

完善并柯里化后的 `withLoadingHoC` 如下：

```

const isEmpty = (prop) => (
  prop === null ||
  prop === undefined ||
  (prop.hasOwnProperty('length') && prop.length === 0) ||
  (prop.constructor === Object && Object.keys(prop).length === 0)
);

const withLoadingHoC = (loadingProp) => (WrappedComponent) => {
  return class extends Component {
    render() {
      return isEmpty(this.props[loadingProp]) ?
        <div className="loader" />
        :
        <WrappedComponent {...this.props}/>;
    }
  }
}

```


在上述代码中实现了 `isEmpty` 函数。因为在复用的情况下，不能再单纯判断数组长度，`this.props[loadingProp]`也可能是对象类型等，所以要通过一个工具函数 `isEmpty` 来判断数据的可用性。当然，`loadingProp`完全可以被设计得更加通用些，如设计成函数形式，由调用者决定何时展现 `loading`，根据返回的布尔值进行判断。

这就是一个基本的高阶组件的设计方案，需求明确，思路清晰。但是高阶组件的“威力”不止于此，现在我们需要在使用 `withLoadingHoC` 的所有情况下，`WrappedComponent` 组件都输出加载数据的具体耗时。实现这个需求很简单，只需在与 `withLoadingHoC` 高阶组件相关的生命周期函数中加入对时间的计算和记录即可。

```
componentDidMount() {
  this.startTimer = Date.now();
}

componentDidUpdate(nextProps) {
  if(!isEmpty(nextProps[loadingProp])) {
    this.endTimer = Date.now();
  }
}

render() {
  const myProps = {
    loadingTime: ((this.endTimer - this.startTimer)/1000).toFixed(2),
  };

  // ...
}
```

在 `componentDidMount` 中记录组件实例化的起始时间，在 `componentDidUpdate` 中记录组件更新的前一刻时间，同时在 `render` 方法中做差并记录。那么，该如何在 `WrappedComponent` 中显示差值 `loadingTime` 呢？这就涉及高阶组件的又一种典型用法：给包裹组件传递不相关的属性 `props`。完整代码如下：

```
const withLoadingHoC = (loadingProp) => (WrappedComponent) => {
  return class extends Component {
    componentDidMount() {
      this.startTimer = Date.now();
```

```

    }

    componentDidUpdate(nextProps) {
      if (!isEmpty(nextProps[loadingProp])) {
        this.endTimer = Date.now();
      }
    }

    render() {
      const myProps = {
        loadingTime: ((this.endTimer - this.startTimer)/1000).toFixed(2),
      };

      return isEmpty(this.props[loadingProp]) ? <div className="loader" /> :
      <WrappedComponent {...this.props} {...myProps}/>;
    }
  }
}

```

回到场景实例中，ContactsApp 组件就可以直接渲染页面内容了。代码如下：

```

render() {
  const { loadingTime } = this.props;
  return(
    <div className="contactsApp">
      <SearchBar filterText={this.state.filterText}
        onUserInput={this.handleUserInput} />
      <ContactList contacts={this.props.contacts}
        filterText={this.state.filterText}/>
      <p>Loading time {loadingTime} seconds</p>
    </div>
  )
}

```

有一些高阶组件的编写会采用 ES 7 的修饰器（decorator）特性：

```

@withLoadingHoC('contacts')
class ContactsApp extends Component {
  // ...
}

```

事实上这和 `withLoadingHoC('contacts')(ContactsApp)` 语句的作用是完全相同的，只不过它更加精简、现代化。

本节我们通过实例剖析了高阶组件，从理论到应用，希望读者能够细心体会。

6.2 高阶组件和 render prop

Michael Jackson 是活跃在 React 社区的一个很有影响力的开发者，他对关于 React 组件复用的问题提出了一个新的方案，并命名为“render prop”。

本节我们就对这个复用方案进行分析，同时对比常用的高阶组件方式。除此之外，再回顾一下 React 解决复用问题的 Mixins 方案的演进过程。

6.2.1 Mixins 的问题

首先，让我们回顾一下 2015 年，广泛使用 `React.createClass` 创建组件的情况。然后设定一个场景：应用需要跟踪鼠标的位置信息，并将其坐标显示在页面上。从那个时期一路走来的开发者，一定不难理解这样的代码：

```
import React from 'react'
import ReactDOM from 'react-dom'

const App = React.createClass({
  getInitialState() {
    return { x: 0, y: 0 }
  },

  handleMouseMove(event) {
    this.setState({
      x: event.clientX,
      y: event.clientY
    })
  },

  render() {
    const { x, y } = this.state
```

```
    return (  
      <div onMouseMove={this.handleMouseMove}>  
        <h1>The mouse position is {x}, {y}</h1>  
      </div>  
    )  
  }  
})
```

在 `React.createClass` 声明的组件中，对 `onMouseMove` 事件进行监听处理，并通过 `setState` 更新鼠标信息。

我们的需求进一步扩大：另一个组件也需要跟踪并显示鼠标的位置信息。最简单的方法就是直接复制、粘贴上面的主要逻辑，但是这样做显然无法达到复用的目的。

为此，在使用 `React.createClass` 创建组件时期，对于代码复用我们通过 Mixins 来实现。关于 Mixins 方式，现在基本已经退出了历史舞台，如果读者不感兴趣，则可以越过相关表述，直接进入 6.2.3 节进行学习。

我们需要创建一个 `MouseMixin`，这样任何一个组件都可以实现跟踪鼠标位置信息代码的复用。`MouseMixin` 代码如下：

```
import React from 'react'  
import ReactDOM from 'react-dom'  
  
const MouseMixin = {  
  getInitialState() {  
    return { x: 0, y: 0 }  
  },  
  
  handleMouseMove(event) {  
    this.setState({  
      x: event.clientX,  
      y: event.clientY  
    })  
  }  
}
```

在消费层面使用 `MouseMixin` 时，代码如下：

```
const App = React.createClass({
  // 使用 MouseMixin
  mixins: [ MouseMixin ],

  render() {
    const { x, y } = this.state

    return (
      <div onMouseMove={this.handleMouseMove}>
        <h1>The mouse position is ({x}, {y})</h1>
      </div>
    )
  }
})
```

这样依靠 Mixins 的思想，代码简捷，思路清晰，相对理想地解决了复用的问题。

6.2.2 高阶组件和 Mixins

随着时代的发展，JavaScript 和 React 技术也加速进化。在 ES 6 标准中增加了 class 特性，React 也做出了相应的回应，引入了更新的组件声明方式。

然而，使用 class 方式声明的组件并不支持 Mixins 特性。并且，使用 Mixins 特性真的是一种完美的解决复用问题的方式吗？其实不是，Mixins 在设计上是存在一些弊端的，具体如下。

- 不确定性。Mixins 会直接修改 state，这就造成了在业务组件中 state 的不确定性。尤其是当一个组件中嵌入了不止一个 Mixins 时，便会经常出现 state 的修改来源不确定的情况。
- 命名冲突。这种情况同样出现在一个组件中嵌入了不止一个 Mixins 的场景中，如果两个组件同时修改了某个 state，那么就会造成冲突。在这种情况下，使用 React.createClass 创建组件时虽然会给出警告，但是只会提醒开发者对此产生警惕，并不能实际有效地解决这个问题。

为了更完美地实现代码复用，在 React 社区中很快出现了高阶组件的方式。这种方式会“修饰”现有的组件，以达到逻辑复用。在同样的场景下，可以通过高阶组件的方式实现复用。

```
import React from 'react'
import ReactDOM from 'react-dom'
```

```
const withMouse = (Component) => {
  return class extends React.Component {
    state = { x: 0, y: 0 }

    handleMouseMove = (event) => {
      this.setState({
        x: event.clientX,
        y: event.clientY
      })
    }

    render() {
      return (
        <div onMouseMove={this.handleMouseMove}>
          <Component {...this.props} mouse={this.state}/>
        </div>
      )
    }
  }
}

const App = React.createClass({
  render() {
    // 将鼠标信息由 state 改为 props
    const { x, y } = this.props.mouse

    return (
      <div>
        <h1>The mouse position is ({x}, {y})</h1>
      </div>
    )
  }
})

const appWithMouse = withMouse(App);
```

为了方便对比，这里同样使用了 `React.createClass` 来创建组件。

我们对原有的 App 组件进行包装，返回一个基于 App 组件的新组件 `appWithMouse`。这个新组件将通过新注入的 `props` 获取到鼠标位置信息。

这种方式很快流行起来。但是我们要停下来思考：这种方式对于 Mixins 的缺点来说，改善了哪些问题呢？我们分开来看。

- 不确定性。这种方式并没有解决不确定性问题，只不过将 `state` 的不确定性转嫁到了 `props` 上。关于一些 `props` 的修改，我们无法区分修改是来自组件内部，还是来自高阶组件的封装过程。
- 命名冲突。命名冲突这个问题依然存在。如果两个高阶组件同时命名并扩充了一个新的 `prop`，那么将依然存在被覆盖的问题。

因此，使用 `class` 方式声明组件并采用 `HoCs` 方法，仍然存在问题，甚至会引入一些静态组合（非动态组合）、更多的样板代码等。

6.2.3 render prop 是什么

本节我们将介绍一种全新的复用方式，即 `render prop`。如果需要对 `render prop` 下一个定义，那么就是：在调用组件时，引入一个函数类型的 `prop`，这个 `prop` 定义了组件的渲染方式。

换句话说，与其使用 Mixins，或者接收并返回一个组件的高阶组件，还不如在正常使用的情况下增加一个 `prop` 来实现在消费层面对不同渲染情况的自定义，最终实现代码复用。

下面我们看看具体如何解决跟踪鼠标位置信息的问题。

```
import React from 'react'
import ReactDOM from 'react-dom'
import PropTypes from 'prop-types'

class Mouse extends React.Component {
  static propTypes = {
    render: PropTypes.func.isRequired
  }

  state = { x: 0, y: 0 }
```

```
handleMouseMove = (event) => {
  this.setState({
    x: event.clientX,
    y: event.clientY
  })
}

render() {
  return (
    <div onMouseMove={this.handleMouseMove}>
      {this.props.render(this.state)}
    </div>
  )
}
}

const App = React.createClass({
  render() {
    return (
      <div>
        <Mouse render={({ x, y }) => (
          // 在这个 render 函数中，可以使用坐标信息
          <h1>The mouse position is ({x}, {y})</h1>
        )}/>
      </div>
    )
  }
})
```

上面代码的关键就在于 Mouse 组件，其维护了内部状态用来计算鼠标位置信息，同时在实例化时，通过调用名为 render 的 prop 属性函数实现了自定义渲染内容的效果。

6.3 React 组件的组合和复用——Function as Child Component

除前面介绍的使用高阶组件及 render prop 方式实现复用和组合外，还有 Function as Child Component 方式。

顾名思义，Function as Child Component 是指父组件接收一个函数以实现复用。代码如下：

```
import { Parent } from './components';

function example() {
  return (
    <Parent>
      { param => <div> {param} </div> }
    </Parent>
  )
}
```

这种方式的特点在于 Parent 组件往往拥有一些内部状态或者需要做一些复杂且共享的计算，这些数据需要对外暴露以实现复用。通过传递函数参数的方式来实现数据复用，对于前端开发者来说并不陌生。

在 Parent 组件的 render 方法中，可以使用 this.props.children 完成渲染。执行 this.props.children 方法，并传递必要参数，即可实现基于共享数据的不同渲染逻辑，从而达到复用的目的。

```
this.props.children(param);
```

我们来看一个简单的例子。假设有一个 LoggedUser 组件，它可以展现用户名以及相关的其他信息。借助于 Function as Child Component 方式，只需定义 LoggedUser 组件的不同 this.props.children 逻辑即可。

```
import LoggedUser from './LoggedUser';

function example1() {
  return (
    <LoggedUser>
      { username => <div> {username} </div> }
    </LoggedUser>
  )
}
```

设想在另一个文件中，需要根据 username 是否合法来分别展现登录和退出按钮。实现代码如下：

```
import React from 'react';
import {LogoutButton, LoginButton} from './component';

function example2() {
  return (
    <LoggedUser>
      {
        username => (
          username ? <LogoutButton/> : <LoginButton/>
        )
      }
    </LoggedUser>
  )
}
```

根据 `username` 信息，我们就实现了多种页面 UI 的展现。其核心在于使用 `LoggedUser` 组件中的 `render` 方式，我们需要根据 `this.props.children` 进行渲染。代码如下：

```
render() {
  return (
    <div> {this.props.children(this.state.username)} </div>
  )
}
```

与高阶组件相比，这种实现方式能够将 `Parent` 与 `Child` 组件解耦，同时传递函数参数的方式也更加灵活。

下面我们再通过一个更加工程化的例子来体会这种实现方式，可以在本书配套的代码仓库中找到完整代码。

在一个页面应用中，可能有多处都需要用到滚动条滚动位置的具体信息。比如，对于用户使用滚动条滚动到的特定位置，以及特定的数据坐标，不同组件都会做出与之相对应的行为（如懒加载组件等）。于是，我们便可以实现一个 `ScrollPos` 的 `Parent` 组件：

```
import React, {Component} from 'react';

class ScrollPos extends Component {
  state = {
    position: null
  }
```

```
}

componentDidMount() {
  window.addEventListener('scroll', this.handleScroll);
}

componentWillUnmount() {
  window.removeEventListener('scroll', this.handleScroll);
}

handleScroll = (event) => {
  const scrollTop = event.target.scrollingElement.scrollTop;
  this.setState({
    position: scrollTop
  })
}

render() {
  return (
    <div> {this.props.children(this.state.position)} </div>
  )
}
}
```

在 `ScrollPos` 组件生命周期函数 `componentDidMount` 和 `componentWillUnmount` 中，分别添加了对滚动事件的监听和清理逻辑。同时，在滚动鼠标时实时更新 `this.state.position`。关键的一步是在 `render` 方法中调用 `this.props.children`，并将 `this.state.position` 作为参数传递，以执行消费层面不同的渲染逻辑。

接下来，所有使用到滚动条数据信息的组件，都可以按照下面列举的方式进行应用。

场景 1：根据滚动位置的不同，显示不同的内容形式。

```
import React, {Component} from 'react';
```

```
class App extends Component {
  render() {
    return (
```

```
    <div className="App">

      <ScrollPos>
        {
          position => <h1> {'Awesome !'.substr(0, position * 15)} </h1>
        }
      </ScrollPos>

      <div className="spacer" />
    </div>
  );
}
```

场景 2: 直接显示出位置信息。

```
class App extends Component {
  render() {
    return (
      <div className="App">

        <ScrollPos>
          {
            position => <h1> { position } </h1>
          }
        </ScrollPos>

        <div className="spacer" />
      </div>
    );
  }
}
```

运行结果如图 6-5 所示。



图6-5 展现滚动坐标效果图

ScrollPos 组件 `props.children` 为函数类型，即：

```
position => <h1> { position } </h1>
```

并且具备 `state.position` 状态，任何需要根据 `position` 数值做出的不同渲染效果便都可以实现，这就是数据复用性的体现。

这种模式同样在 React 相关的“轮子”设计中有广泛的应用，比如 `react-motion` 等。这种模式也存在弊端：对于使用 `this.props.children()` 渲染的组件部分，我们难以直接使用 `shouldComponentUpdate()` 来进行性能优化。因为在每次渲染周期中都声明了一个新的函数，在使用 `shouldComponentUpdate` 进行 `props` 浅比较时，都会永远返回 `true`。

在实际的工程环境中，还需要开发者根据项目设计做出合理的选择。

6.4 React 组件的组合和复用——Children API

本节我们将介绍使用 React Children API 实现组件的组合和复用，并最终实现一个可复用的轮播图组件。

首先回顾一下 `this.props.children` 属性。在前面章节中我们介绍过它有多种取值，比如在 `Parent` 组件中：

```
import React, {Component} from 'react';
class Parent extends Component {
  render() {
    return (
      <div>
```

```
        {this.props.children}
      </div>
    )
  }
}
```

在消费层面，如果 Parent 组件没有子节点，它就是 undefined:

```
import React, {Component} from 'react';
import Parent from '../Parent';
```

```
class App extends Component {
  render() {
    return (
      <div className="App">

        <Parent></Parent>

      </div>
    );
  }
}
```

在 Parent 组件中，this.props.children 为字符串的情况:

```
class App extends Component {
  render() {
    return (
      <div className="App">

        <Parent>
          hello
        </Parent>

      </div>
    );
  }
}
```

在 Parent 组件中存在一个节点，此时 this.props.children 数据类型是 object:

```
class App extends Component {
  render() {
    return (
      <div className="App">
```

```

    <Parent>
      <p> single p </p>
    </Parent>
  )
}
}

```

存在多个子节点，此时 `this.props.children` 数据类型是 `array`：

```

class App extends Component {
  render() {
    return (
      <div className="App">

        <Parent>
          <p> Multiple p </p>
          <p> Multiple p </p>
          <p> Multiple p </p>
          <p> Multiple p </p>
        </Parent>
      )
    )
  }
}

```

另外，React 提供了一个顶层方法 `React.Children` 来处理 `this.props.children` 的不同取值情况。我们可以用 `React.Children.map` 来遍历子节点，用 `React.Children.only` 来保证子节点的单一性等。

我们来看一个典型的例子。

```

import React, { Component, Children } from 'react';

class Parent extends Component {
  render() {
    const buttons = Children.map(this.props.children, child => (
      <button>
        {child}
      </button>
    ))
  }
}

```

```

return (
  <div>
    <h1>Total Children: { Children.count(this.props.children) }</h1>
    {buttons}
  </div>
)
}
}

```

上述代码通过 `Children.map`，实现了将业务代码传入的 `this.props.children` 渲染为 `button` 组件，并通过 `Children.count` 方法计算出传入的 `this.props.children` 的数目并加以展现。

那么如何利用这些特性来实现组合和复用，并实现轮播效果呢？事实上，执行上面的代码，便可以实现一个轮播图的雏形。想象一下这样的场景：`buttons` 组合就是轮播焦点图的切换 `icon`，或者将 `buttons` 换成 `Image` 标签就是不同的焦点图。常见的一种轮播效果如图 6-6 所示。



图6-6 轮播效果图

是不是有所启发？我们再来看一个实际例子。

```

import React, { Component } from 'react';
import SlideShow from './SlideShow';

class App extends Component {
  render() {
    return (
      <div className="App">
        <SlideShow>
          
          

```

```
        
      </SlideShow>
    </div>
  )
}
```

在上述业务代码中调用了 SlideShow 组件,三张图片作为 SlideShow 组件的子节点出现。

SlideShow 组件应该具有 this.state.total 和 this.state.current 两个状态值,分别代表总的轮播图数目和当前轮播图的序号。同时需要定时实现切换的逻辑脚本,这可以借助于 setInterval 来实现,请参见如下代码。

```
import React, { Component, Children } from 'react';
```

```
class SlideShow extends Component {
```

```
  state = {
    total: 0,
    current: 0
  }
```

```
  componentDidMount() {
    const { children } = this.props;
    this.setState({ total: Children.count(children) });
    this.interval = setInterval(this.showNext, 3000);
  }
```

```
  componentWillUnmount() {
    clearInterval(this.interval);
  }
```

```
  showNext = () => {
    const { total, current } = this.state;
    this.setState({
      current: current + 1 === total ? 0 : current + 1
    });
  }
```

```
render() {  
  const { children } = this.props;  
  const bullets = Array(this.state.total).fill("o");  
  bullets[this.state.current] = "●";  
  
  return (  
    <div className="slideshow">  
      <div>{bullets}</div>  
      {Children.toArray(children) [this.state.current]}  
    </div>  
  )  
}
```

在 `componentDidMount` 组件生命周期函数中进行 `this.state.total` 值的计算，并加入轮播定时器：每 3 秒执行一次 `this.showNext` 方法。`this.showNext` 方法所需要做的事情很简单，就是更新 `this.state.current` 值。真正的切换效果，需要配合样式文件来实现。

为了达到轮播渐变的效果，最终代码可以使用 `react-addons-css-transition-group` 动画库：

```
import ReactCSSTransitionGroup from 'react-addons-css-transition-group';
```

为了配合实现轮播图翻阅的动画效果，需要在 `render()` 方法中添加以下代码：

```
<ReactCSSTransitionGroup  
  className="group"  
  transitionName="example"  
  transitionEnterTimeout={800}  
  transitionLeaveTimeout={800}  
>  
  {Children.toArray(children) [this.state.current]}  
</ReactCSSTransitionGroup>
```

`ReactCSSTransitionGroup` 是 React 官方推荐的用于实现过渡动画的组件，它基于 `ReactTransitionGroup`。事实上，`ReactTransitionGroup` 会在 `children` 数量发生变化时调用对应的钩子方法，实现渐变动画。

综上所述，结合 `React.Children` 顶层 API，搭配使用 `this.props.children` 实现复用的原理，我们实现了一个简单的轮播图组件。读者可在本书配套的代码仓库中找到本节的完整代码。

6.5 React “轮子”是怎样设计的

通过前面章节的介绍，我们学习到一些复用技巧。本节我们将继续深化，从零开始，从需求出发，实现一个简单的 React 模态框组件——“轮子”。“轮子”作为软件工程中的俚语，指的是经过良好抽象封装后能够多次复用、迅速上手，并且在一定程度上支持自定义设计的库。

在前端开发中，UI 组件“轮子”非常流行，比如 ant-design 就是非常优秀的基于 React 的 UI 库，其中包含了模态框的实现。本节示例力求简单、清晰，其中贯穿抽象设计和复用知识。

接下来，我们将分析三种“轮子”的实现，这三种设计将会从“简单、粗暴”开始，逐渐接近 react-modal 库的思想，循序渐进，由浅入深。

大部分网站都需要用到模态框交互。事实上，模态框就是我们俗称的“弹窗”，只不过比下面的基本实现多了更多的信息承载和交互行为。

```
alert('我是一个简单、原生的 alert~');
```

同时，为了使网站页面更加美观和吸引眼球，模态框在设计上往往会搭配深色透明的遮罩，如图 6-7 所示。

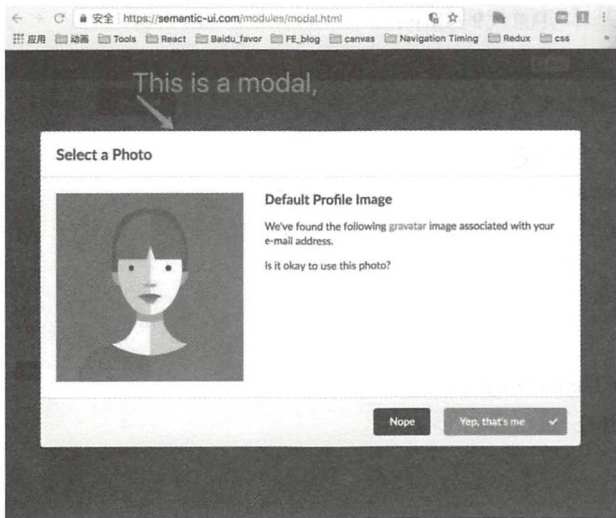


图6-7 模态框效果

我们常见的用户登录框、错误信息提示框等，都是非常典型的模态框。

在传统的 jQuery 操作 DOM 类库的技术栈下，我们可以“肆无忌惮”地选择 DOM 节点，

完成添加、移除等操作，实现模态框并不复杂。可是在 React 和 Redux 世界里，该如何实现模态框呢？

组件设计如图 6-8 所示。其中需要格外注意触发模态框组件和模态框组件的层级关系。

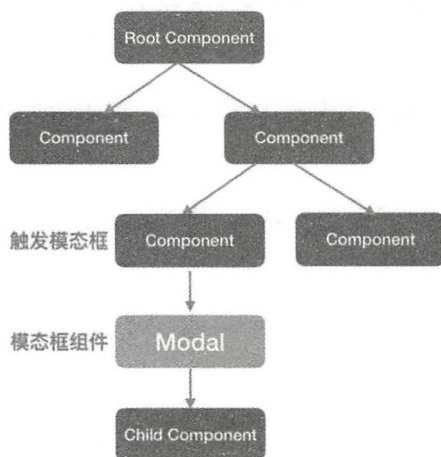


图6-8 组件设计图

该组件树对应的基本页面代码如下：

```
export default class App extends Component {
  render() {
    <div className="app">
      <div className="left">
        <h1>Hello left</h1>
        // ...
      </div>

      <div className="right">
        <h1>Hello right</h1>
        // ...
        <div>
          <BadModal>
            // 模态框内容
            <h1> Modal title </h1>
            <p> Modal content</p>
          </div>
        </div>
      </div>
    </div>
  }
}
```

```

        </BadModal>
      </div>
    </div>
  </div>
}
}

```

因为每个模态框里的内容和交互不尽相同，所以在 `BadModal` 组件内我们可以根据上文介绍过的 `Child Component` 进行复用，渲染不同的内容。这个 `Child Component` 将会由消费层根据弹窗的不同业务需求开发完成，最终实现模态框内容及交互的复用设计。代码如下：

```

class BadModal extends Comment {
  render() {
    return (
      <div className="modal">
        { this.props.children }
      </div>
    )
  }
}

```

至此，我们就实现了最基本的模态框。但是这种方法比较原始、简陋，不完美的地方还有很多。打开样式表文件：

```

body .modal {
  position: fixed;
  // ...
}

.left {
  z-index: 3
}

.right {
  z-index: 1
}

```

我们会发现恼人的 `z-index` 问题，模态框是 `.right` 节点的子孙节点，而 `.right` 的 `z-index` 小于 `.left` 的 `z-index`，这样造成的直接问题就是模态框不能脱离整体页面而“突出显示”。

细想一下，这个问题的根本原因在于模态框组件的层级设置有误。观察图 6-8，因为很深层次的子孙组件是模态框的触发源，如果模态框作为其子孙组件出现，这样就很难覆盖于整体页

面之上。

想想我们平时使用的 jQuery 是怎么做的：

```
$('body').append('<div class="overlay"></div>');
```

一般情况下，模态框和遮罩总是作为 body 的第一层子节点出现的。于是，我们很自然地想到：只需改变模态框组件出现的位置即可。这就需要模态框组件和触发模态框出现的深层次组件进行通信，使得模态框组件能够取得触发组件的数据信息，并加以展示。

传统的 React 组件间通信无外乎通过 props 和基于 props 的回调实现（暂不考虑 context 的“黑魔法”）。可是这样做太过复杂，也难以实现复用，更不利于维护。

更好的设计，还要从调整后的页面组件树结构出发。如图 6-9 所示，在 document.body 下设计加入了 Modal 组件，与 Root Component 并列。同时，在触发模态框的组件下加入了一个 Fake Modal 组件。事实上，Fake Modal 组件并不渲染任何结果，而是借助于生命周期函数，完成在 document.body 下新建并插入 Modal 组件的使命。

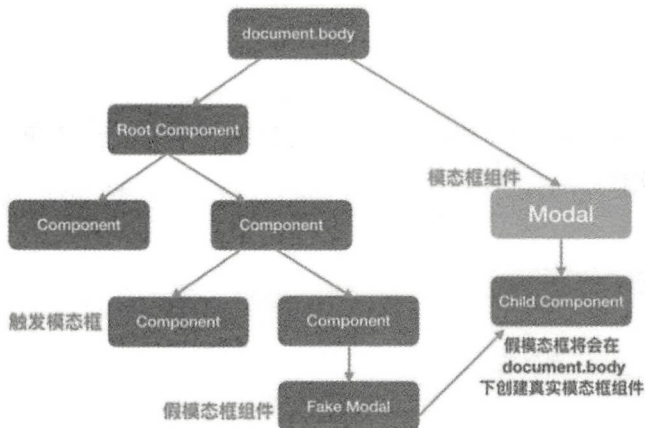


图6-9 调整后的组件设计图

下面借助于代码进行理解。

```
class Modal extends Component {
  componentDidMount() {
    this.modalTarget = document.createElement('div');
    this.modalTarget.className = 'modal';
    document.body.appendChild(this.modalTarget);
    this.renderModal();
  }
}
```

```

    }
    componentWillUpdate() {
      this.renderModal();
    }
    componentWillUnmount() {
      ReactDOM.unmountComponentAtNode(this.modalTarget);
      document.body.removeChild(this.modalTarget);
    }
    renderModal() {
      ReactDOM.render(
        <div>{ this.props.children }</div>,
        this.modalTarget
      );
    }
    render() {
      return <noscript />
    }
  }
}

```

在真正的 `render` 方法中，不渲染任何实质的内容，而是 `return <noscript />`，或者直接 `return null`。

同时，借助于生命周期函数 `componentDidMount`，使用原生 JavaScript 实现在 `body` 下创建模态框。

```

this.modalTarget = document.createElement('div');
this.modalTarget.className = 'modal';
document.body.appendChild(this.modalTarget);

```

最后调用 `renderModal` 方法完成向页面根节点插入模态框的操作。

```

this.renderModal();

ReactDOM.render(
  <div>{ this.props.children }</div>,
  this.modalTarget
);

```

相信很多 React 开发者都会使用 `Redux` 来进行数据管理。分析图 6-9 所示的结构，这样的实现对 `Redux` 的友好兼容并不到位。

比如，如果在 Modal 组件的子组件 Child Component 中需要使用 Redux store 里的数据，那么因为 Provider 不在 Modal 组件的组件链中，所以 Child Component 无法感知 Redux store 的存在。具体问题如图 6-10 所示。

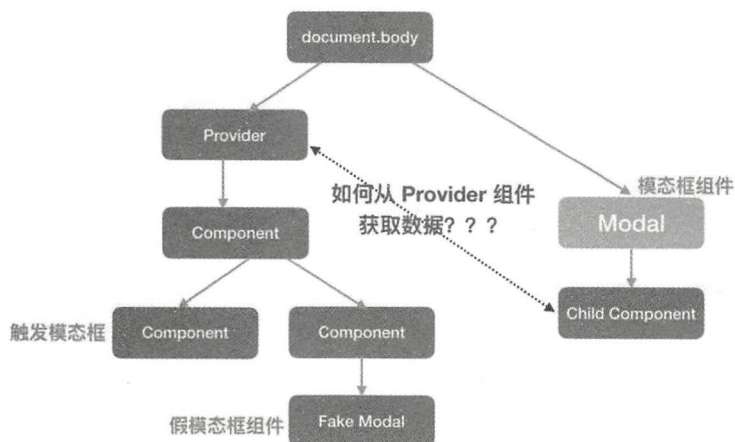


图6-10 组件设计中的问题

为了解决这个问题，我们继续改进组件树结构，如图 6-11 所示。

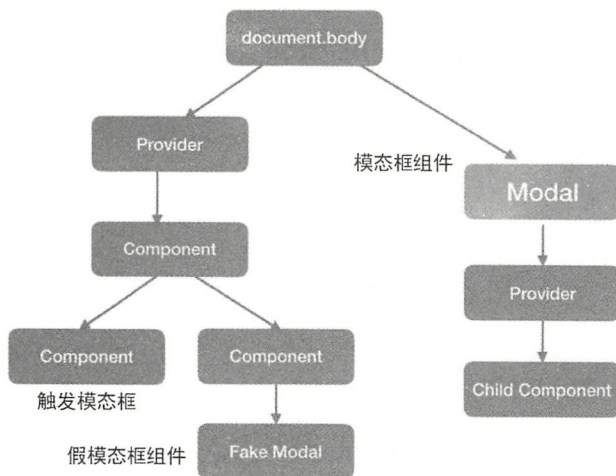


图6-11 改进后的组件设计图

为此，我们引入应用的 store，以及 react-redux 库提供的 Provider 组件。


```
import { store } from '../index';
import { Provider } from 'react-redux';
```

同时，改动前面的 `renderModal` 方法，加入对 `Provider` 的支持。

```
renderModal() {
  ReactDOM.render(
    <Provider store={ store }>
      <div>{ this.props.children }</div>
    <Provider>,
    this.modalTarget
  );
}
```

在 React 开发中，很多工程师对 `react-modal` 都非常熟悉。很多开发者都依赖它来使用模态框。这个库设计良好，且封装完善。事实上，本节介绍的内容完全体现了 `react-modal` 的设计思路。

另外，React 16 带来的新特性 `Portal` 与本节内容密切相关——我们把 `Portal` 形象地翻译为“传送门”。`Portal` 允许将组件渲染到父节点之外的 DOM 节点。对 `Portal` 的基本使用如下：

```
render() {
  return ReactDOM.createPortal(
    this.props.children,
    anyDomNode,
  );
}
```

这里 React 并不会在当前结构中渲染组件，而是向 `anyDomNode` 中渲染 `this.props.children`，`anyDomNode` 是指任何有效的 DOM 节点，而不管它处于哪个层级位置。

我们能够使用此特性来简化上文中的逻辑。打开 `react-modal` 最新提交的源码，便能够发现对这个新特性的支持。在 `react-modal/src/components/Modal.js` 文件中有：

```
const isReact16 = ReactDOM.createPortal !== undefined;
const createPortal = isReact16
  ? ReactDOM.createPortal
  : ReactDOM.unstable_renderSubtreeIntoContainer;
```

这里对 React 版本进行判断，并通过了 `isReact16` 标志位来设置是否支持 `createPortal` 方法。在不支持的情况下，使用 `unstable_renderSubtreeIntoContainer` 这个不稳定的 API，它类似于

`createPortal`，将组件更新到所传入的 DOM 节点中，这里使用它完成了跨组件的 DOM 操作。

最后在 `render` 方法内可以明显地看到，对于不支持 `createPortal` 的情况，采用了类似的设计逻辑；否则就使用 `createPortal` 方法。

```
render() {
  if (!canUseDOM || !isReact16) {
    return null;
  }

  if (!this.node && isReact16) {
    this.node = document.createElement("div");
  }

  return createPortal(
    <ModalPortal
      ref={this.portalRef}
      defaultStyles={Modal.defaultStyles}
      {...this.props}
    />,
    this.node
  );
}
```

6.6 setState 异步带来的讨论和思考

在 React 组件中，调用 `this.setState()` 是最基本的场景。这个方法通过状态变化来触发组件更新，同时这个看似平常的 `this.setState()` 方法里面也许蕴含了很多精巧的设计。

很多开发者都已经意识到，`setState` 方法“或许”是异步的。看上去更新状态是如此轻而易举，那么为什么要设计成异步的呢？实际上，状态更新会触发组件重新渲染，而隐藏在重新渲染背后的代价更大。如果出现在短时间内反复渲染的情况，那么在性能考量上是不可取的。所以，React 可能会采用批量更新策略，集合一系列连续的状态更新，而最后只触发一次渲染。

在所有场景下 `setState` 方法都是异步的吗？在 React 中又是如何实现异步的？有没有方法规

避 `setState` 的异步过程呢？本节就将揭晓答案。

6.6.1 `setState` 的同步和异步之争

为了回答上面的问题，我们试图从 React 官方来寻找确切说法：

“`setState()` does not always immediately update the component. It may batch or defer the update until later. This makes reading `this.state` right after calling `setState()` a potential pitfall.”

官方表示，`setState()`并不会保证更新状态的同步性。虽然官方说法相对来说比较笼统，但我们还是能从 `batch later` 或者 `defer` 这样的用词中读出来：一定存在“异步”的更新情况。`batch` 可翻译为“批处理”，`defer` 又表示“延期、推迟”之意。同时要注意，在这两个动词之前，`may` 又表现出这样的“异步”并不代表适用于所有 `state` 更新的情况，仿佛一些状态更新又会是同步的。

那么在 `setState` 方法的背后发生了什么？其实这是一个很复杂的过程，React 从最初的版本到现在的版本，中间也经历了很多次修改。它的工作除了直观更改 `this.state` 的值之外，还要负责触发重新渲染逻辑，这里面要经过 React 核心 `diff` 算法，最后才能决定是否要进行重渲染，以及如何渲染。事实上，`setState` 将根据不同的情况来决定是把变化存入一个临时队列中，还是执行队列中的更新。换句话说，多个 `setState` 调用的情况有可能被合并，因此出于性能的考量，`this.setState()` 设计以延迟的方式来执行是很合理的。

社区中有一个非常著名的实验，用来验证 `this.setState()` 的同步和异步问题，代码如下：

```
function incrementMultiple() {  
  this.setState({count: this.state.count + 1});  
  this.setState({count: this.state.count + 1});  
  this.setState({count: this.state.count + 1});  
}
```

直观上看，当 `incrementMultiple` 函数被调用时，组件状态的 `count` 值被增加了 3 次，每次增加 1，最后 `count` 值增加了 3。但是，运行代码并分析输出会发现，最终只是 `state` 增加了 1。

考虑如下代码：

```
componentDidMount() {  
  document.querySelector('#btn-row').addEventListener('click', this.onClick);  
}  
  
onClick() {  
  this.setState({count: this.state.count + 1});  
}
```

```
    console.log('# this.state', this.state);
  }
  // .....
  render() {
    console.log('#enter render');
    return (
      <div>
        <div>{this.state.count}
        <button id="btn-row">Increment Row</button>
      </div>
    </div>
    )
  }
}
```

运行上面的代码，我们能得到准确的 count 值。

深入源码，在 React 的 `setState` 函数实现中，会根据 `isBatchingUpdates` 变量判断是直接更新 `this.state` 还是放到队列中稍后更新。`isBatchingUpdates` 的默认值是 `false`，表示 `setState` 会同步更新 `this.state`。但是存在一个函数 `batchedUpdates`，这个函数会把 `isBatchingUpdates` 修改为 `true`，而 React 在调用事件处理函数之前就会调用 `batchedUpdates`。这样造成的后果就是在由 React 控制的事件处理过程中，`setState` 不会同步更新 `this.state`，而是将记录了 `newState` 的组件存入 `dirtyComponent` 中，以便后续更新。

我们可以通过图 6-12 来进行了解。

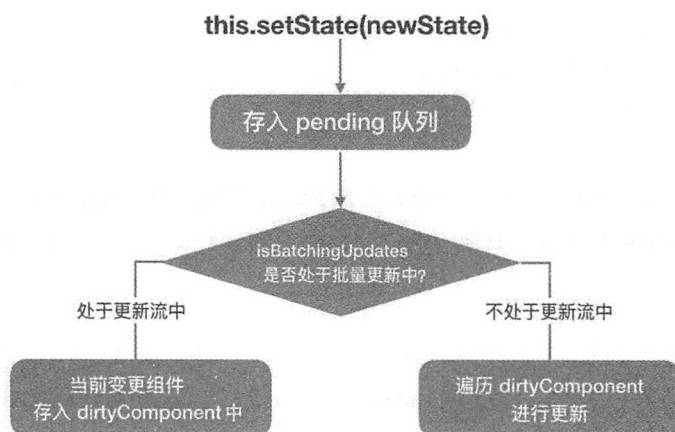


图6-12 setState更新机制示意图

从图 6-12 中可以看出，这里的判断和分支导致了 `setState` 方法的不同行为。

总结：在由 React 控制的事件处理过程中，`setState` 不会同步更新状态，而在 React 控制之外，`setState` 会同步更新 `state`。那么如何理解 React 控制之内外呢？在大多数使用情况下，在交互事件过程中使用的都是在 React 库中处理、封装的事件，例如 `select`、`input`、`button` 等，我们认为处于 React 库的控制之下的。在这种情况下，`setState` 就会以异步的方式执行。

实际上，绕过 React 通过 JavaScript 原生 `addEventListener` 直接添加的事件处理函数，在使用 `setTimeout/setInterval` 等 React 无法掌控的 API 时，就会出现同步更新状态的情况。在上面代码中，我们使用了 `addEventListener`，因此触发了同步更新，得到了正确的 `count` 值。

为此，Redux 作者 Dan Abramov（已经加入 React 团队）给出了说法：

“In current release, they will be batched together if you are inside a React event handler. React batches all `setStates` done during a React event handler, and applies them just before exiting its own browser event handler.”

根据此说法，在 React 事件处理控制内外，调用 `this.setState()` 的方法不尽相同。在 React 事件处理控制之内，将会有 `state` 的合并过程，因此状态的改变是异步发生的；在 React 事件处理控制之外，将会同步变更 `state`。

6.6.2 让 `setState` 连续更新的方法

如果想保证同步更新或者连续更新该如何做呢？对于如下代码：

```
this.setState({count: this.state.count + 1});
this.setState({count: this.state.count + 1});
this.setState({count: this.state.count + 1});
```

如果想让 `count` 值一次性加 3，解决方法如下。

方法一：将一个回调函数传入 `setState` 方法中，即著名的函数式用法，这样能保证即使更新被收集（batched）时，也能访问到预期的状态或 `props`。

```
setState((prevState, props) => ({
  count: prevState.count + 1
}));
setState((prevState, props) => ({
  count: prevState.count + 1
```



```
    ));  
    setState((prevState, props) => ({  
      count: prevState.count + 1  
    }));  
  });
```

方法二：把 `setState` 更新之后的逻辑（比如上述代码中连续第二次 `count+1`）封装到一个函数中，并作为第二个参数传给 `setState`。这个函数逻辑将会在更新后由 React 代理执行。即：

```
this.setState(updater, [callback]);
```

也就是说，`setState` 函数的第二个参数允许传入回调函数，该函数在更新完毕后会调用执行。

```
this.setState({count: this.state.count + 1}, () => {  
  this.setState({count: this.state.count + 1}, () => {  
    this.setState({count: this.state.count + 1})  
  });  
});
```

方法三：把 `setState` 更新之后的逻辑放在一个合适的生命周期函数中，比如 `componentDidMount` 或者 `componentDidUpdate`。也就是说，第一次 `count+1` 之后，触发 `componentDidUpdate` 生命周期函数，将第二次 `count+1` 操作直接放在 `componentDidUpdate` 函数中即可。

这种方法有效地利用了 React 组件的生命周期函数，实质上跟 `setState` 的用法并没有太大关系。

6.6.3 `setState` Promise 化的讨论和尝试

在 JavaScript 中处理异步流程，很流行的一种做法是使用 `Promise`，那么能否应用这个思路改造 `setState` 方法呢？

说具体一些，就是在调用 `setState` 方法之后，立即返回一个 `Promise` 对象，状态更新完毕后可以调用 `promise.then` 进行下一步处理。

如果 `setState` 方法返回的是一个 `Promise` 对象，自然会更加优雅，为此发起者这样解释：

“`setState()` currently accepts an optional second argument for callback and returns undefined. This results in a callback hell for a very stateful component. Having it return a promise would make it much more manageable.”

即 `setState` 可以接收第二个参数，该参数为函数类型，将会作为异步更新后的回调函数。

这样的使用方式虽然保证了 state 的正常更新和准确性，但是很容易造成回调地狱现象的出现。如果 setState() 方法可以返回一个 Promise 对象，将会极大方便对状态的控制。但是这样做真的可以被 React 接受，改变 React 吗？答案是否定的。

React 官方认为：解决异步带来的困扰有很多方案。比如可以在合适的生命周期钩子函数中完成相关逻辑（见前面的方法三）。另外，React 官方也考虑到了未来调整的问题。现有的批量更新策略，还是有一些问题的。React 团队在后期也许会进行调整，但是在批量更新策略被调整之前，盲目地扩充 setState 接口只会被认为是一种短视的行为。

对此，Redux 作者 Dan Abramov 也发表了自己的看法。他认为，任何需要使用 setState 的第二个参数 callback 的场景，都可以使用生命周期函数 componentDidUpdate（或者 componentDidMount）来达到目的，这样完全可以规避回调地狱问题。

笔者认为，在 JavaScript 社区中，确实有很多第三方库渐渐地接受并使用了 Promise 风格，但是这些库解决的问题往往都是强异步性的，比如文件读取、网络操作等。React 似乎没有必要增加这个新特性。另外，如果每个 setState 都返回一个 Promise 对象，也会带来性能问题。总而言之，为了解决 setState 异步带来的问题，直接让 setState 返回一个 Promise 对象是画蛇添足的。

但是，作为学习者和 React 生态中的一员，我们有必要了解并参与到这样的设计讨论当中。这对于加深 React 理解，提高开发者的设计能力都非常有帮助。甚至，即使 React 官方不接受 setStatePromise 化的特性，我们也可以在自己的业务中完成封装和支持。

```
setStateAsync(state) {  
  return new Promise((resolve) => {  
    this.setState(state, resolve)  
  });  
}
```

我们使用 Promise 包装原生的 this.setState 方法为 setStateAsync 方法，setStateAsync 返回一个 Promise 对象。关于 setState 的设计，充分体现了 React 的思想，也引起了广泛讨论。了解这些内容，对于理解 React 会更上一个台阶。

6.7 React 组件和 React element 到底是什么

我们了解了 React 组件，可能也听说过 React element，或者对 React.createElement 方法也不陌生。可是它们之间到底是什么关系呢？本节我们将进行梳理，同时为下一节的 React 实现进

行知识储备。

首先思考 React 到底是什么。

React 是一个构建视图层的框架。无论 React 本身如何复杂，也不管其生态如何庞大，构建视图始终是它的核心。

接下来讲讲 React element。

简单地说，React element 描述了用户在屏幕上看到的事物。抽象地说，React element 是一个描述了 DOM 节点的对象。

请注意这里的用词“描述”，因为 React element 并不是屏幕上看见的真实事物，而是一个描述真实事物的集合。那么 React element 的存在有什么意义呢？为什么会有 JavaScript 对象描述 DOM 节点的设计呢？

事实上，JavaScript 对象是轻量级的，用对象来作为 React element，React 可以轻松地创建或销毁这些元素，而不必太担心操作成本。同时，React 具有分析这些对象的能力，也就具有分析虚拟的 DOM 节点的能力。当 DOM 结构需要改变时，相比于直接更新真实的 DOM，更新虚拟的 DOM 性能优势非常明显。

为了创建描述 DOM 节点的对象(或者 React element)，我们可以使用 `React.createElement` 方法。

```
const element = React.createElement(  
  'div',  
  {id: 'login-btn'},  
  'Login'  
)
```

在这里 `React.createElement` 方法接收三个参数：

- 一个表述标签名称的字符串（div、span 等）。
- 当前 React element 需要具有的属性。
- 当前 React element 要表达的内容，或者一个子元素。

`React.createElement` 方法被调用之后，会返回一个 JavaScript 对象，即 React element。

```
{  
  type: 'div',
```

```
props: {  
  children: 'Login',  
  id: 'login-btn'  
}  
}
```

接下来，当使用 `ReactDOM.render` 方法时，才会渲染到真实的 DOM 中。

```
<div id='login-btn'>  
  Login  
</div>
```

此时，才得到一个真实的 DOM 节点。

进一步设想，在真正开发时，如果不直接使用 `React.createElement` 来创建 React element，无疑是极其烦琐的。这时候就出现了 React Component，即 React 组件。

React 组件就是一个函数或者一个 Class，它根据输入参数最终返回一个 React element，而不需要我们直接手写 React element。

所以说，实际上使用 React 组件，React 组件借助于 `React.createElement` 方法，进而生成 React element，这对于开发体验的提升无疑是巨大的。这个过程可用图 6-13 来描述。

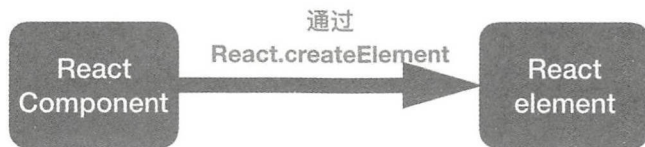


图6-13 `React.createElement`示意图

请看如下代码：

```
function Button ({ onLogin }) {  
  return React.createElement(  
    'div',  
    {id: 'login-btn', onClick: onLogin},  
    'Login'  
  )  
}
```

这里定义了一个 Button 组件,它接收 onLogin 参数,并返回一个 React.createElement 函数,此函数最终返回一个 React element。注意: onLogin 参数是一个函数,并且像 id:'login-btn'一样成为这个 React element 的属性。

我们看到了一个 React element 类型为 HTML 标签(“span”、“div”)的情况。事实上,我们也可以传递另一个 React element 类型,代码如下:

```
const element = React.createElement(  
  User,  
  {name: 'Lucas'},  
  null  
)
```

注意:此时 React.createElement 的第一个参数是另一个 React element,这与类型为 HTML 标签的情况不尽相同。当 React 发现类型为一个 Class 或者函数时,它就查询这个 Class 或者函数会返回什么样的 element,并为这个 element 设置正确的属性。

类似于递归,React 会不断地重复这个过程,直到不再存在“createElement 调用类型为 Class 或函数”的情况。

我们结合代码再来体会一下。

```
function Button ({ addFriend }) {  
  return React.createElement(  
    "button",  
    { onClick: addFriend },  
    "Add Friend"  
  )  
}  
  
function User({ name, addFriend }) {  
  return React.createElement(  
    "div",  
    null,  
    React.createElement( "p", null, name ),  
    React.createElement(Button, { addFriend })  
  )  
}
```

上面有两个组件: Button 和 User。User 组件描述 DOM 是一个 div 标签,在这个 div 内,

又存在一个 `p` 标签，这个 `p` 标签展示了用户的 `name`，同时它还存在一个 `Button` 组件。

现在我们来看在 `User` 和 `Button` 中 `React.createElement` 的返回情况，输出 `React.createElement` 的返回结果。

```
function Button ({ addFriend }) {
  return {
    type: 'button',
    props: {
      onClick: addFriend,
      children: 'Add Friend'
    }
  }
}

function User ({ name, addFriend }) {
  return {
    type: 'div',
    props: {
      children: [{
        type: 'p',
        props: { children: name }
      },
      {
        type: Button,
        props: { addFriend }
      }
    ]
  }
}
```

在上面的输出中，我们看到有 4 个 `type` 值：'button'、'div'、'p'和 `Button`。

当 `React` 发现 `type` 值是 `Button` 组件时，它就查询这个 `Button` 组件会返回什么样的 `React element`，并赋予正确的 `props`。

最终 `React` 会得到完整的表述 `DOM` 树的对象。在这个例子中，就是：

```
{
  type: 'div',
```

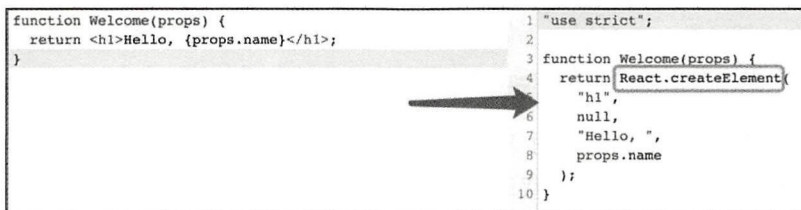
```
props: {
  children: [{
    type: 'p',
    props: { children: 'Tyler McGinnis' }
  },
  {
    type: 'button',
    props: {
      onClick: addFriend,
      children: 'Add Friend'
    }
  }
]}
}
```

在编写 React 组件时，可能大家都在使用 JSX 来描述虚拟的 DOM。事实上，JSX 总是被编译成 `React.createElement` 而被调用。一般 Babel 为我们做了 `JSX→React.createElement` 这件事情。当然，React 也可以脱离 JSX 而存在。

再来看一个例子，代码如下，其编译结果如图 6-14 所示。

```
function Button ({ addFriend }) {
  return React.createElement(
    "button",
    { onClick: addFriend },
    "Add Friend"
  )
}

function User({ name, addFriend }) {
  return React.createElement(
    "div",
    null,
    React.createElement( "p", null, name),
    React.createElement(Button, { addFriend })
  )
}
```

```
function Welcome(props) {  
  return <h1>Hello, {props.name}</h1>;  
}  
  
1 "use strict";  
2  
3 function Welcome(props) {  
4   return React.createElement(  
5     "h1",  
6     null,  
7     "Hello, ",  
8     props.name  
9   );  
10 }
```

图6-14 编译结果

对应于 JSX 用法，就是大家熟悉的：

```
function Button ({ addFriend }) {  
  return (  
    <button onClick={addFriend}>Add Friend</button>  
  )  
}  
  
function User ({ name, addFriend }) {  
  return (  
    <div>  
      <p>{name}</p>  
      <Button addFriend={addFriend}/>  
    </div>  
  )  
}
```

最后，本书实现了一个小工具（在本书配套的代码仓库中）对 JSX 进行实时编译。编译平台一分为二，左边写 JSX，右边实时展现其编译结果，如图 6-15 所示。

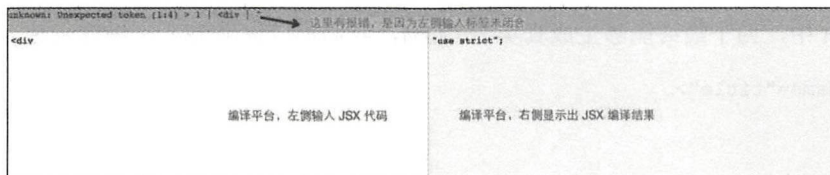


图6-15 编译平台

感兴趣的读者可以研究其实现，十分简单。

6.8 实现一个简易的 React 库

使用 React 进行开发，也许开发者只需要了解编写组件、清楚 props 和状态、明白动态调用

setState 和生命周期函数，就可以上手做项目了。学习 React 也许比学习 Angular.js 等类库更简单一些，但是要想开发出一个好的 React 应用，一定需要了解更多的内容。

有了上一节的知识积累，我们对 React 的理解又相对深入一些。接下来，我们不妨尝试开发一个简易的 React 库，相信在编写代码的过程中，很多概念都会融会贯通，也会更加理解 React 的设计理念和 workflows。

回到最原始的前端开发时期，DOM 是 W3C 组织推荐的用于处理 HTML 的标准编程接口。在网页上，页面（或文档）的对象被组织在一个树形结构中，其中用来表示文档中对象的标准模型就称为 DOM，如图 6-16 所示。

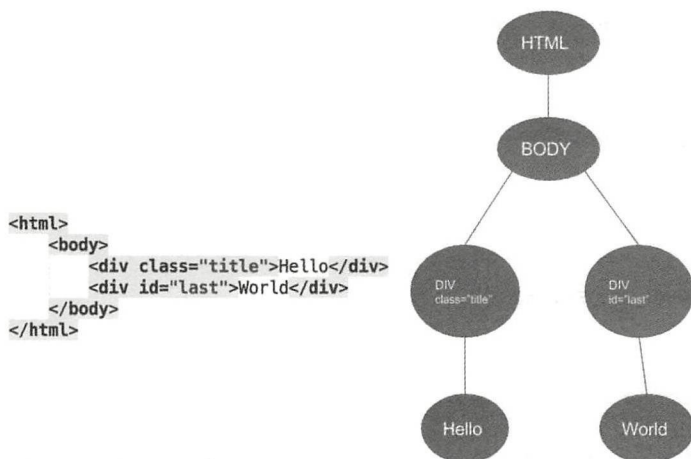


图6-16 DOM示意图

在 React 中，为了最终能够生成真实的 DOM，我们可以使用 JSX 来进行表达，代码如下：

```
<div className="title">
  hello
</div>
```

经过编译之后得到：

```
React.createElement(div, {className: 'title'}, 'hello');
```

也可以这样编写无状态组件：

```
const simpleComponent = () => {
  return <h1>Hello World</h1>;
}
```

使用 Babel 编译之后，就得到：

```
const mySimpleComponent = () => {  
  return React.createElement('h1', null, `Hello World`);  
}
```

更复杂的一个 JSX 样例如下：

```
<div style={{color: yellow}}>  
  <Title text='I'm a title' />  
  <Person name='moki' />  
</div>
```

```
<div style={{color: yellow}}>  
  <Title text="I'm a title" />  
  <Person name="moki" />  
</div>
```

经过编译之后得到以下结果：

```
React.createElement(  
  "div",  
  { style: { color: yellow } },  
  React.createElement(Title, { text: "I'm a title" }, null),  
  React.createElement(Person, { name: "moki" }, null)  
);
```

依赖 JSX 和 React 的关系，我们了解了 React 的 createElement 方法，接下来就开始实现一简易的 React 库。在社区中对 React 的模拟层出不穷，为了节省篇幅，同时达到梳理知识点的目的，我们采用 Ofir Dagan 的实现。

6.8.1 准备工作

首先编写一个宿主 HTML 文件，在这个文件中引入 react.js 和业务 app.js。

```
<!DOCTYPE html>  
<html>  
  <head>  
    <title>Build Your Own React</title>  
  </head>  
  <body>
```

```
<div id="root"></div>
<script src="./src/react.js"></script>
<script src="./src/app.js"></script>
</body>
</html>
```

app.js 直接使用 `React.createElement` 和 `ReactDOM.render` 方法:

```
const helloWorld = React.createElement('div', null, `Hello World`);
ReactDOM.render(helloWorld, document.getElementById('root'));
```

为了使上述业务代码能够正常工作,我们在自己的 `react.js` 文件中进行 API 模拟。`react.js` 文件的内容如下:

```
function anElement(element, children) {
  const anElement = document.createElement(element);
  anElement.innerHTML = children.join(' ');
  return anElement;
}

function createElement(el, props, ...children) {
  return anElement(el, children);
}

window.React = {
  createElement
};

window.ReactDOM = {
  render: (el, domEl) => {
    domEl.appendChild(el);
  }
};
```

事实上,这里的 `createElement` 方法的第一个参数只支持 HTML 节点,它将调用 `anElement` 函数,通过 `document.createElement(element)` 创建简单的 DOM 节点。在 `ReactDOM.render` 函数中,使用 `appendChild` 对新生成的 DOM 节点进行挂载。

从完整的实现来看,除支持 HTML 节点以外,还需要支持无状态组件(即函数式组件),以及使用 `class` 声明的组件。下面我们将一一进行处理

6.8.2 初见雏形

首先处理对无状态组件的支持。对无状态组件的使用，请参考 `app.js` 文件内容：

```
const Hello = function () {  
  return React.createElement('div', null, `Hello World`);  
};  
const helloWorld = React.createElement(Hello, null, null);  
ReactDOM.render(helloWorld, document.getElementById('root'));
```

现在需要做的就是 在 `react.js` 文件 中对 `React.createElement` 方法进行兼容：当 `element` 参数是一个函数，即为无状态组件时，直接进行调用即可。在 `react.js` 文件中：

```
function anElement(element, children) {  
  if (typeof(element) === 'function') {  
    return element();  
  }  
  else {  
    const anElement = document.createElement(element);  
    anElement.innerHTML = children.join(' ');  
    return anElement;  
  }  
}
```

再深入考虑，`children` 也不仅仅是简单的文本节点，它还可能是其他子组件。在 `app.js` 文件中：

```
const Hello = function () {  
  return React.createElement('div', null, `Hello World`);  
};  
const helloWorld = React.createElement(Hello, null, null);  
const helloWorld2 = React.createElement(Hello, null, null);  
const regularDiv = React.createElement('div', null, `I'm just a regular div`);  
  
const parent = React.createElement('div', null,  
  helloWorld,  
  helloWorld2,  
  regularDiv,  
  ` I'm just a text`  
);
```

针对这种情况，我们需要对 `react.js` 文件中的 `anElement` 方法的参数 `children` 进行遍历。对于非文本节点的情况，需要进行递归调用。在 `react.js` 文件中：

```
function anElement(element, children) {
  if (typeof(element) === 'function') {
    return element();
  }
  else {
    const anElement = document.createElement(element);
    children.forEach(child => {
      if (typeof(child) === 'object') {
        anElement.appendChild(child);
      }
      else {
        anElement.innerHTML += child;
      }
    });
    return anElement;
  }
}
```

至此，`React.createElement` 方法支持接收 DOM 节点和函数这两种类型的参数。当然，还支持 `class` 组件类型的参数。在 `app.js` 文件中：

```
class Hello {
  render() {
    return React.createElement('div', null, `Hello World`);
  }
}

const helloWorld = React.createElement(Hello, null, null);
ReactDOM.render(helloWorld, document.getElementById('root'));
```

为此，我们需要针对 `class` 的情况进行判断。如果命中 `class`，则直接实例化并返回其 `render` 方法。在 `react.js` 文件中：

```
function anElement(element, children) {
  if (isClass(element)) {
    const component = new element();
```



```
    return component.render();
  }
  else {
    // all the rest
  }
}
```

我们使用以下方式判断 class 的情况：

```
function isClass(func) { return typeof func === 'function' && /^class/.test(Function.
prototype.toString.call(func)); }
```

现在是时候重新整理一下代码了。anElement 函数已经变得很庞大，为此我们将逻辑进行分离。具体做法是新建一个 react-utils.js 文件，用于存放一些工具方法，如 isClass、isStatelessComponent 等。

6.8.3 持续迭代

目前，我们已经实现了渲染 DOM 节点，接收无状态组件和 class 组件。下面要考虑的就是为这些组件加入 props 和 state。在 app.js 文件中：

```
const Hello = ({name}) => {
  return React.createElement('div', null, `Hello ${name}`);
};
const helloWorld = React.createElement(Hello, {name: 'Ofir'}, null);
ReactDOM.render(helloWorld, document.getElementById('root'));
```

我们给 Hello 组件添加了名为 name 的 props，对于无状态组件来说，处理起来很简单，只需要加入 props 参数即可。在 react.js 文件中：

```
if (isStateLessComponent(element)) {
  return element(props);
}
```

对于 class 组件的 props 情况，在 app.js 文件中处理如下：

```
class Hello extends React.Component {
  constructor(props) {
    super(props);
  }

  render() {
```

```
    return React.createElement('div', null, `Hello ${this.props.name}`);
  }
}

const helloWorld = React.createElement(Hello, {name: 'Ofir'}, null);
ReactDOM.render(helloWorld, document.getElementById('root'));
```

Hello 组件继承自 `React.Component`，为了实现 `extends React.Component` 调用，我们需要创建一个 `Component` 父类，在此类中进行 `props` 的初始化和赋值。

```
class Component {
  constructor(props) {
    this.props = props;
  }
}
```

```
window.React = {
  createElement,
  Component
};
```

同时针对这种情况，我们采用 `handleClass` 方法进行处理。

```
if (isClass(element)) {
  return handleClass(element, props);
}
```

更加复杂的情况是，比如想实现一个 `MyButton` 组件，点击按钮后触发弹出框。在 `app.js` 文件中：

```
class MyButton extends React.Component {
  constructor(props) {
    super(props);
  }

  render() {
    return React.createElement('button', {onclick: this.props.onClick}, `Click me`);
  }
}
```

```
const myBtn = React.createElement(MyButton, {onClick: () => alert('yay it worked')}, null);
ReactDOM.render(myBtn, document.getElementById('root'));
```

针对这种情况，我们需要对其属性进行遍历，并且对事件回调类型（即以 on 开头）的属性进行相应的事件绑定，使用原生 `addEventListener` 方法：

```
function handleHtmlElement(element, props, children) {
  //...

  Object.keys(props).forEach(propName => {
    if (/^on.*$/.test(propName)) {
      anElement.addEventListener(propName.substring(2).toLowerCase(), props[propName]);
    }
    else {
      anElement.setAttribute(propName, props[propName]);
    }
  });
  return anElement;
}
```

`handleHtmlElement` 函数变得庞大起来，我们将提取出 `appendChild` 和 `appendProp` 函数。同时，针对操作对象或数组的情况（`Object.keys`、`forEach`），我们使用 `lodash` 类库进行处理。

当然，对 `props` 处理之后，还需要对 `state` 进行处理。

为了更清晰地表述，这里实现一个计数应用：页面上有两个按钮，点击其中一个按钮计数加 1，点击另外一个按钮则计数减 1。在 `app.js` 文件中：

```
class Counter extends React.Component {
  constructor(props) {
    super(props);
    this.state = {value: 0};
  }

  onPlusClick() {
    this.setState({value: this.state.value + 1});
  }

  onMinusClick() {
    this.setState({value: this.state.value - 1});
  }
}
```

```

    }

    render() {
      return React.createElement('div', null,
        React.createElement('div', null, `The Famous Dan Abramov's Counter`),
        React.createElement('div', null, `${this.state.value}`),
        React.createElement('button', {onClick: this.onPlusClick.bind(this)}, '+'),
        React.createElement('button', {onClick: this.onMinusClick.bind(this)}, '-')
      );
    }
  }

```

为了让上述代码正常运行，我们需要开发 `setState` API。这里将使用一种粗暴的算法：每次调用 `setState` 时，都将对组件进行暴力“重新渲染”，就是先删除老节点，再构建新节点。在 `react.js` 文件中：

```

class Component {
  // same code as before...
  setState(state) {
    this.state = Object.assign({}, this.state, state);
    reRender();
  }
}

function reRender() {
  while (rootDOMElement.hasChildNodes()) {
    rootDOMElement.removeChild(rootDOMElement.lastChild);
  }
  ReactDOM.render(rootReactElement, rootDOMElement);
}

```

我们给 `Component` 增加了 `setState` 方法，该方法使用 `Object.assign` 对状态进行更新，然后调用 `reRender` 函数进行重新渲染，并使用 `removeChild` 对老节点进行清除。更新做法如下：

- 清除原有的 DOM 节点。
- 渲染新的根节点。
- 渲染所有的子节点。

这样的做法是极其粗暴的，实际上可以进行一定程度的优化处理。比如，在重新渲染发生时，每个 `class` 组件及其 `state` 都重新生成一遍，这是非常不合理的。为了解决这个问题，则可

以这样做：

- 对已经实例化的 class 进行缓存。
- 当处理一个已经在缓存池中的 class 时，直接返回缓存池中的 class 实例。
- 使用标志位对 React class 进行标识。
- 当挂载子节点时，如果标志位表示这是一个 React class，那么直接调用其 render 方法。

因此，在 handleClass 中：

```
function handleClass(clazz, props, children) {  
  classCounter++;  
  if (classMap[classCounter]) {  
    return classMap[classCounter];  
  }  
  const reactElement = new clazz(props);  
  reactElement.children = children;  
  reactElement.type = REACT_CLASS;  
  classMap[classCounter] = reactElement;  
  return reactElement;  
}
```

在 handleChild 中：

```
if (child.type === REACT_CLASS) {  
  appendChild(element, child.render());  
}
```

除此之外，更好的算法应该表现为：

- JavaScript 环境比较完善。
- 相比于直接创建 DOM 节点，不如使用 JavaScript 对象来表示虚拟的 DOM 树。
- 对于新的渲染更新诉求，生成新的 JavaScript 对象表示虚拟的 DOM 树。
- 计算出两棵虚拟的 DOM 树的 diff。
- 将 diff 集更新到真实的 DOM 中。

到此为止，我们已经实现了简易的 React 库，基于该库也开发了一个经典的 Todo list 应用。

其代码在本书配套的代码仓库中可以找到，感兴趣的读者可以进一步了解。相信通过这个简易 React 库的编写，各位读者能够了解到实现成熟 React 应用的流程。

6.8.4 总结与思考

本节的目的不仅仅是实现一个生产可用的 React 应用，更在于基于此将 React 组件的声明、编译、生命周期等典型知识点串联起来。

仔细思考我们实现的 React 应用，会发现这个应用其实还是比较低效的。如同前面所说的，每次调用 `setState` 时都会清除相关的 DOM 节点，并从头进行构建。另外，React 的亮点就是对虚拟 DOM 的实现以及 diff 算法，然而在实现这个应用时，并没有涉及高效的 diff 算法。

因此，为了更真实地模拟 React 的工作，至少还需要：

- 通过 JavaScript 对象模拟虚拟的 DOM。
- 使用 `render` 方法对虚拟的 DOM 进行操作，而不是直接应用在真实的 DOM 中。
- 收集 diff，并计算出应该对真实的 DOM 所做的最小更新。
- 将最小更新应用在真实的 DOM 中。

关于计算 diff 的细节，React 通过大胆假设和前提策略，将两棵树的比较时间复杂度由 $O(n^3)$ 降低为 $O(n)$ 。

具体表现为：

- 在前端页面中，DOM 节点跨层级的移动操作特别少，可以忽略不计。
- 拥有相同类型的两个组件将会生成相似的树形结构，拥有不同类型的两个组件将会生成不同的树形结构。
- 对于同一层级的一组子节点，它们可以通过唯一 id 进行区分。

基于第一点，React 对树的比较算法，实际上只对树进行分层比较，两棵树只会对同一层次的节点进行比较。这样只需要对树进行一次遍历，便能完成整棵 DOM 树的比较。基于第二点，React 在进行 diff 时，如果发现对比两项是同一类型的组件，则按照原策略继续比较；如果类型不同，则直接进行替换，而不再对组件下的子节点进行比较。基于第三点，React 列表节点组件通过开发者设置的唯一 key，来协助实现添加、删除和排序的操作。

当然,React 允许开发者通过 `shouldComponentUpdate` 方法来直接决定组件是否需要进行 diff。

事实上,上述过程在 React 中叫作 reconciliation。

另外,React 最新的 reconciliation 引擎是 Fiber,隐藏在 Fiber 之后的思想非常有趣:旧的 reconciliation 过程是线性的、同步的,即一旦 reconciliation 过程开始就不能中断。Fiber 却不一样,它打破了阻塞设计。下面我们来看一个场景,体会 Fiber 带来的益处。

比如有一个文本节点需要更新,同时页面中正在执行一个非常复杂的动画。在需求上,首先要保证动画的流畅运行,否则用户将明显感觉到页面卡顿。而对文本节点的更新,延迟几毫秒感觉并不是很明显。为此,Fiber 建立了一种中断机制,在 reconciliation 过程中,React 都可以控制 Fiber 的运行与暂停。暂停时,表示需要进行其他计算或渲染(比如展现动画),之后再继续进行 reconciliation 过程,这样就会使得应用的运行更加流畅顺滑。

当然其实现非常复杂,本节不再展开描述。

6.9 引入 Redux 的必要性及利弊

对于准备或者已经开始使用 React 的开发者来说,考虑是否需要 Redux 来进行状态管理,是非常必要而又令人困惑的,甚至有的初学者抱怨:因为 Redux 的原因,使其对 React 望而却步、心生畏惧。本节我们就来看 Redux 的本质,以及在项目中引入 Redux 的利弊。

首先,我们不能因为“绝大多数 React 应用都采用了 Redux 进行状态管理”而盲目地应用,这显然是一种从众心理。你是否想过:React 依靠组件状态及组件间通信机制,其实已经完成了所谓的状态管理。那为什么还需要 Redux 呢?

让我们从数据流开始说起。我们已经很清楚 props 传递和单向数据流的含义:数据通过 props 在组件间层层传递,并通过基于 props 的回调实现组件间通信。

我们来看一个实际例子,非常简单的计数器,如图 6-17 所示。

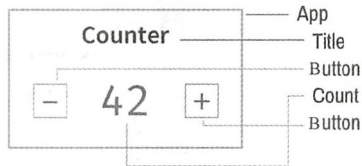


图6-17 计数器示意图

将当前计数 count 存储在 App 组件的 state 当中,并通过 prop 传递给显示计数的 Count 组件,如图 6-18 所示。

对于数据(即当前计数 count)的更新,是顺着组件树“逆流而上”的,如图 6-19 所示。

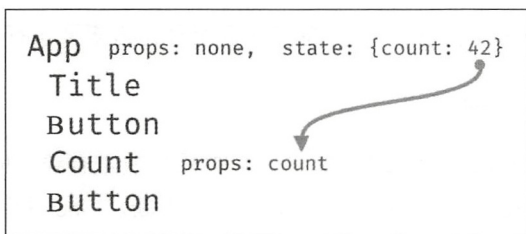


图6-18 props传递示意图

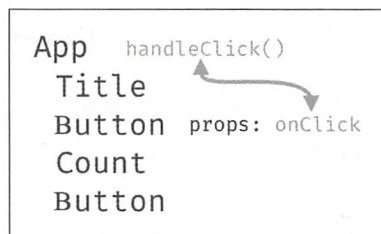


图6-19 点击响应示意图

Button 组件响应点击事件，触发 App 组件内的 handleClick 函数执行。为此，App 组件通过 props 传递 handleClick 给 Button 组件，来完成双方的通信过程。

我们想象一下，在这样一个简单却很典型的 React 应用中，数据就像电流一样，通过各色导线在组件之间川流不息。同时，导线不可能凭空产生电流，它的两端一定连接着组件。这些组件有的是发电机，产生数据；有的是用电器，消费数据。

但是对于一些更复杂的应用，组件层级也会变得更加复杂。比如图 6-20 所示的 Twitter 页面，页面中有三处 Avatar 组件都需要展示用户的基本信息（用户头像存储在用户基本信息当中）。

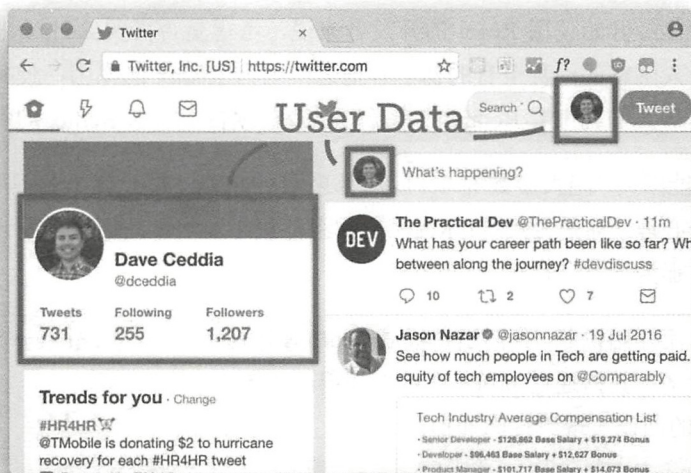


图6-20 Twitter页面

合理的做法是把用户信息（user）存储在所有关心此项数据的组件的共同父组件当中。举例来说，如果 App 组件维护了用户信息（user），为了将用户信息传递到相关三处的深层组件当

中，则需要做出这样的设计，如图 6-21 所示。

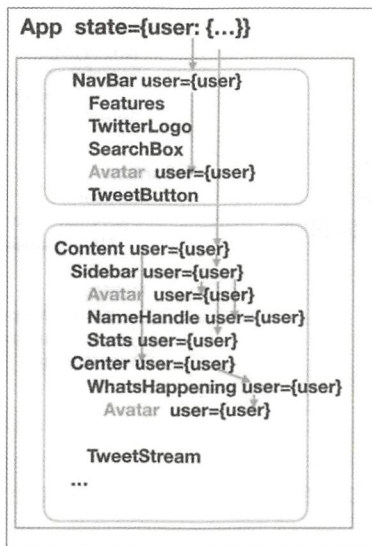


图6-21 组件间通信设计图

上述设计基于 React 原则，“导线”不至于拧成一团“乱麻”，因为所有的数据流向都是单向的，非常清晰。

但存在的问题是，为了将用户信息（user）传递给 Avatar 组件，很多不必要的中间层组件都需要承接 user 数据并向下传递，从而导致这些中间层组件很难保持“职责单一”的特性，进而很难被复用。比如，为了将 user 数据传递给第二处 Avatar 组件，App 组件需要将 user 数据传递给 Content 组件，再由 Content 组件传递给 Sidebar 组件，最后由 Sidebar 组件传递给最终的消费数据组件——Avatar 组件。在整个过程中，Content 和 Sidebar 组件本身并不关心 user 数据。

对于这些中间层组件来说，如果能做到“不需要这些数据，就不用看见或传递这些数据”不应该是更好的设计吗？

Redux 便能实现类似的需求，解决相应的问题。借助于 connect 函数，我们可以给任何需要感知 store 内容的组件注入 store 中的 state 片段以及相关方法。

如图 6-22 所示，这就是 Redux 存在的意义之一。

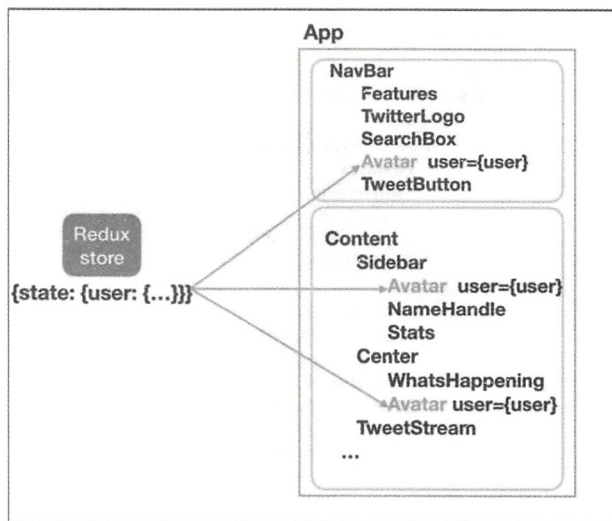


图6-22 在Redux架构下组件间通信设计图

除此之外，Redux 还具备很多优秀的特性，比如调试的灵活性、代码的易维护性、时间旅行特性（一种撤销功能）等，它们都使得应用架构能得到更好的维护和扩展。

当然，使用 Redux 也存在一些限制，主要体现在如下几个方面。

- Redux 带来了函数式编程、不可变性思想等理念。为了遵循这些理念，开发者必须写很多“模式代码（boilerplate）”，烦琐以及重复是开发者不愿容忍的。当然也有很多 hack 旨在减少样板模式，但目前可以说 Redux 天生就带着烦琐。
- 应用需要使用 object 或者 array 描述状态。
- 应用需要使用 plain object 类型的 action 来描述变化。
- 应用需要使用纯函数来处理变化。

以上种种限制，使得开发者很难痛痛快地编写业务代码，一旦发生任何变化，就要对应地编写 action（action creator）、reducer 等，这样一来，和偏响应式的解决方案如 MobX 相比，编程体验会打折扣。另外，Redux 可以理解为一个简易的发布订阅系统，因此带来的内存消费也需要额外考虑。

当然，从另一个角度来看，这些限制都会转化成闪光之处，缺点又变成了优点：

- 便于调试。

- 便于线上错误收集，只需要发送 state、action 等快照即可。
- 方便结合 localStorage 初始化 store。
- 便于服务端渲染。
- 便于开发在线协作型应用。
- 方便实现时间旅行特性（Undo/Redo 功能）。
- 便于测试。

总之，Redux 带来了无与伦比的调试便利性和开发体验，也带来了无法避免的样板代码和重复劳动。

通过本节介绍，希望能使读者深入理解 Redux 的数据管理架构，尤其是对 connect 函数能有一个感性上的认知，并对是否将 Redux 引入到现有的架构中有一个更加准确的判断。

6.10 如何设计并应用 Redux connect

通过前面介绍，我们了解到在使用 Redux 的情况下，使用 connect 进行组件连接，返回获得数据的高阶组件。但是在纷杂的组件树中，如何使用 connect 做出更好的设计，这对于开发和维护体验都至关重要。

react-redux 库提供的 connect 方法非常轻便，但初学者容易陷入的一个误区就是滥用 connect，而没有进行更合理的设计分析。也可能只在顶层进行了 connect 设计，然后再一层层进行数据传递。

比如在一个页面中存在 Profile、Feeds（信息流）、Images（图片）区域，如图 6-23 所示。

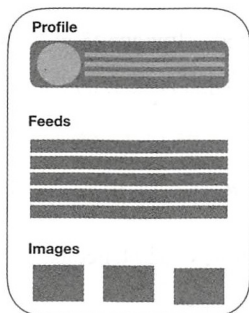


图6-23 页面示意图

这些区域构成了页面的主体，它们分别对应于 Profile、Feeds、Images 组件，共同作为 Page 组件的子组件而存在。

```
<Page>
  <Profile/>
  <Feeds/>
  <Images/>
</Page>
```

如果只对 Page 这个顶层组件进行 connect 设计，其他组件的数据依靠 Page 组件进行分发，则设计如图 6-24 所示。

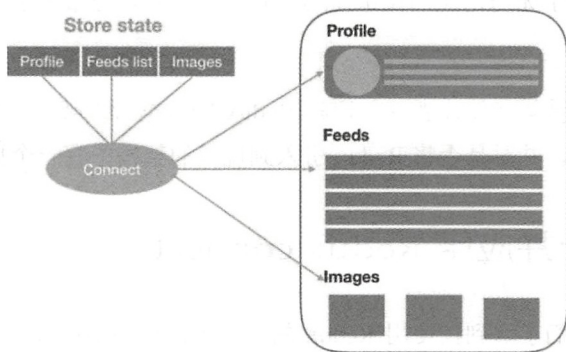


图6-24 connect设计图

这样做存在的问题如下：

- 当改动 Profile 组件中的用户头像时，由于数据变动整个 Page 组件都会重新渲染。
- 当删除 Feeds 组件中的一条信息时，整个 Page 组件也都会重新渲染。
- 当在 Images 组件中添加一张图片时，整个 Page 组件同样都会重新渲染。

因此，更好的做法是对 Profile、Feeds、Images 这三个组件分别进行 connect 设计，在 connect 方法中使用 `mapStateToProps` 筛选出不同组件关心的 state 部分，如图 6-25 所示。

这样做的好处很明显：

- 当改动 Profile 组件中的用户头像时，只有 Profile 组件重新渲染。
- 当删除 Feeds 组件中的一条信息时，只有 Feeds 组件重新渲染。
- 当在 Images 组件中添加一张图片时，只有 Images 组件重新渲染。

这三个组件只连接相关内容，只关心自己需要的数据，这样明显减少了不必要的渲染。

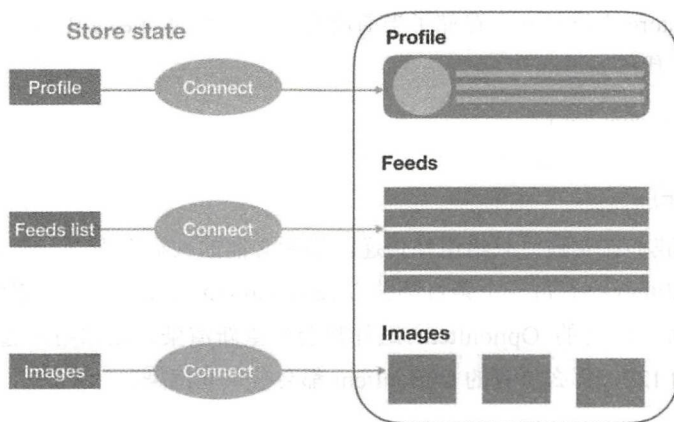


图6-25 对三个组件分别进行connect设计图

实际上，这也是社区中类似于 `reselect` 的类库存在的价值。使用 `reselect` 类库，我们可以定义 `selector`，用于获取 `Redux state` 的一部分，它具有记忆功能，从而规避了不必要的衍生数据的重新渲染和计算过程。

当然，这样的设计在所有的场景下并不都是最佳实践，具体的还要结合自己的业务需求来做。

目前最佳实践是将组件按照“展现（`Presentational`）”或者“容器（`Container`）”进行分类，并在合理的地方抽象出一个进行数据连接的容器组件。

总之，试着在数据流贯通和组件职责间找到平衡才是关键。黄金法则就是：只给组件传递渲染必需的数据。

我们再来看一个例子，体会数据设计的严谨之处。比如需要渲染 300 多个单项选择条目：

```
<Options>
  {this.props.items.map(({ content, itemId }) => (
    <OptionItem
      onClick={selectItem}
      content={content}
      itemId={itemId}
      key={itemId}
    />
  ))}
</Options>
```

当一个条目被点击时，`action` 将会被触发并引起 `store` 的更新，进而该选择条目的样式转变为

已选样式。我们在 store 中维护一个存储了当前所有已选条目的 `selectedItem`，并对每一个条目都进行 `connect` 连接，代码如下：

```
const OptionItem = connect(
  ({ selectedItem }) => ({ selectedItem })
)(SimpleOptionItem);
```

从表面上看，我们在实践“只给组件传递渲染所必需的数据”理念。但真的是每一个条目都需要感知 `selectedItem` 吗？每一个条目都关心 `selectedItem` 造成的后果可想而知：当 store 中的 `selectedItem` 更新时，所有的 `OptionItem` 组件都会被重新渲染。比如用户选择了第 127 项，`selectedItem` 更新为 127，那么所有的 `OptionItem` 都会被重新渲染。

其实，这是一种错误的“只给组件传递渲染所必需的数据”实践。每个选择条目真正关心的不是当前被选中的条目序号，而是“自己有没有被选中”。如果被选中，则做出响应进行样式切换，否则不需要改变。也就是说，当用户选择第 127 项时，`selectedItem` 更新为 127，作为第 125 项需要感知“选中的不是我”，因此“我不需要重新渲染”，而不关心“选中的是谁”。

对应于代码，只需要多传入一个 `prop` 来表示“是否被选中”即可。

```
const OptionItem = connect(
  ({ selectedItem }, { itemId }) => ({ isSelected: selectedItem === itemId })
)(SimpleOptionItem);
```

这就是我们一直强调的，只给组件传递其必需的最小单元数据。

当然，“只给组件传递渲染必需的数据”并不是很容易做到的。有时候我们需要对数据结构进行调整。或者更进一步地说，需要对数据结构进行扁平化处理。扁平化的数据结构是一个很形象的概念，它不仅能够合理引导开发逻辑，同时也是性能优化的一种体现。

请看这样的数据结构：

```
{
  articles: [{
    comments: [{
      authors: [{
      }]
    }]
  }],
  ...
}
```

不难想象这是一个文章列表加文章评论互动的场景，其对应于三个组件：Article、Comment 和 Author。

这样的页面设计比比皆是，如图 6-26 所示。



图6-26 页面效果

上述数据结构很难做到“只给组件传递渲染必需的数据”，这三个组件无一例外都要关心 store 中的 articles 数组，每个组件数据源的变动都会“牵一发而动全身”。同样，对相关 reducer 的处理也很棘手，如果 articles[2].comments[4].authors1 发生了变化，想要返回更新后的状态，并保证不可变性，操作起来不是那么简单的。

因此，更好的数据结构设计一定是扁平化的，我们对 articles、comments、authors 进行扁平化处理。例如 comments 数组不再存储 authors 数据，而是记录 userId，需要时从 users 数组中进行提取即可。

```
{
  articles: [{
```

```
...
  },
  comments: [{
    articleId: ...,
    userId: ...,
    ...
  }],
  users: [{
    ...
  }]
}
```

不同组件只需要关心不同的数据片段，比如 `Comment` 组件只关心 `comments` 数组；`Author` 组件只关心 `users` 数组。这样不仅操作更合理，而且有效减少了渲染压力。

6.11 使用 selector 实现最佳实践

上一节我们提到了 `selector`，它并不是 `Redux` 带来的，在任何版本的 `Redux` 源码中都找不到 `selector` 的定义，开发者自然也不能从 `Redux` 中将它引入进来。但是 `selector` 却十分重要，它往往和 `React`、`Redux` 最佳实践联系在一起。

本节我们就来了解 `selector` 的奥秘。简单地说，`selector` 是一个“方便函数”或者“工具函数”，这个函数用于查询或计算 `Redux store` 的衍生数据。我们知道，在 `Redux store` 中应该仅仅维护应用的基本数据状态。但是在开发中，我们需要查询或组合可用的数据状态，由数据状态计算出衍生数据供组件使用。说到这里，`selector` 就是用来处理“从 `Redux store` 数据到组件渲染所需数据”这一转变过程的。

但是，这样分离处理有什么好处呢？为什么称之为“最佳实践”呢？直观上，`selector` 使得 `Redux` 只需要维护最核心、最基本的数据状态，保证了 `store` 的单一性和纯净。同时，将更多的计算过程用 `selector` 函数处理，实现了推衍计算的灵活性，职责分离，更好地服务于业务代码。此外，借助于 `selector` 相关类库，也可以实现数据的缓存，达到更好的性能。

6.11.1 selector 使用实例

设想我们在设计一个酒店预订系统，当用户预订房间后，需要显示预订信息，如“亲爱的 XX（预订者姓名），您已经成功预订了 XXXX 年 XX 月 XX 日 XXX 酒店 XX（房型）房间。入

住日期: XXXX 年 XX 月 XX 日; 退房日期: XXXX 年 XX 月 XX 日”。

渲染如上信息, 所需要的数据状态着实不少。就拿预订者姓名这一项来说, 自然需要维护预订者的姓名。在实际开发中, 预订成功页面 store 含有 userInfo 等状态, 我们只需要在组件的 render 方法内对数据进行渲染即可。

```
const { name } = this.props;
return (
  // ...
  <p> 预订者姓名: {name} </p>
  // ...
)
```

如果想在姓名前面加上“Mr/Mrs/Miss”这样的称谓, 则可以根据用户的年龄、性别、是否单身等信息计算得到, 代码如下:

```
let title = '';
if (userInfo.gender === 'Male') {
  title = "Mr.";
}
else if (userInfo.gender.maritalStatus === 'Married') {
  title = "Mrs.";
}
else {
  title = "Miss.";
}
```

这样的逻辑放在 Redux 代码的哪个环节合适呢?

首先可以排除放在 render 方法中。因为 render 方法直接影响渲染性能, 所以应尽可能保持简单。那么放在 Redux mapStateToProps 中如何? Redux 暴露的 mapStateToProps 方法理应是一个将 state 映射到 props 的函数, 如果到处都有这样的计算, 显然函数逻辑与函数名 mapStateToProps 并不相符, 也不合适。同时, 如果另外一个组件也需要类似的 title 信息该怎么办? 难道还要复制一段相同的计算代码到该组件的 mapStateToProps 方法中吗? 这显然不是优雅的做法。

事实上, 上述计算逻辑就反映了 selector 函数需要完成的任务。我们可以实现一个 selector 函数 selectUserName, 并进行完善。


```
export const selectUserName = (state) => {
  let title = '';
  if (state.userInfo.gender === 'Male') {
    title = "Mr.";
  }
  else if (state.userInfo.gender.maritalStatus === 'Married') {
    title = "Mrs.";
  }
  else {
    title = "Miss.";
  }

  const userNameWithTitle = `${title} ${state.userInfo.userName}`;
  return userNameWithTitle;
}
```

这样一来，所有相关的 `mapStateToProps` 函数也可以直接使用 `selectUserName` 函数，完成精准数据的推衍。

```
const mapStateToProps = (state) => {
  return {
    // ...
    userNameWithTitle: selectUserName(state)
  }
}
```

6.11.2 使用神奇的 `reselect` 类库

在实际的应用开发中，通常开发者会使用 `reselect` 类库来辅助完成“由 `Redux` store 推导出所需数据”这一过程。那么 `reselect` 具有什么特性呢？

简单地讲，基于 `selector` 思想，`reselect` 类库具有如下几个特性。

- 在 `selector` 函数的基础上，它完成了推衍或计算衍生数据的任务，使得 `Redux` 可以只维护最基本的数据状态。
- 通过使用 `reselect`，使得 `selector` 函数更高效。如果 `selector` 函数的参数不变，则不会重新执行函数体逻辑。

- 通过使用 `reselect`，使得 `selector` 具有组合特性。

为了更好地理解，这里举一个例子。如图 6-27 所示，页面中有“已选分类”和“选择分类”两个模块，用户可以在“选择分类”中选择相关的分类项目，并添加到“已选分类”中。其中，“选择分类”中的所有分类条目往往通过异步请求获取。



图6-27 页面例子

在这样的场景下，通常需要两个相关的 `reducer` 函数：`allItems` 和 `selectedIds`，它们分别用来对所有条目和已选择条目进行处理。

`allItems` 维护的数据类似于：

```
[
  {id: 1, item: {content: "", ...}},
  {id: 2, item: {content: "", ...}},
  {id: 3, item: {content: "", ...}},
  {id: 4, item: {content: "", ...}},
  ...
]
```

`selectedIds` 记录了用户已选择的条目 `id`。注意，为了避免数据冗余，这里并没记录条目的具体信息。

```
selectedIds = [1, 5];
```

这样一来，为了展示用户已选择的条目，开发者需要关心这两个 reducer 函数的组合计算结果，就是需要 selectedIds 从 allItems 的每一个已选 id 中进行筛选，计算得到类似的数据结构：

```
selectedItems = [  
  {id: 1, item: {content: "", ...}},  
  {id: 5, item: {content: "", ...}}  
]
```

以上计算过程示意图如图 6-28 所示。

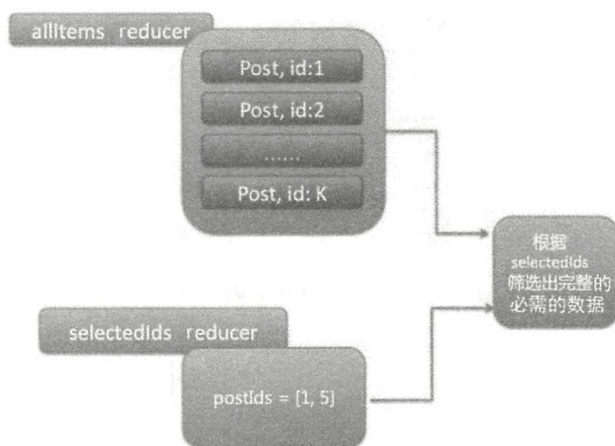


图6-28 计算过程示意图

结合前一节所讲的知识，这个计算过程的实现代码如下：

```
const selectedItems = props.allItems.filter(item => selectedIds.includes(item.id))
```

这样的 selector 实现存在如下几个问题。

- 需要相关的 React 组件知晓必要的数据结构，比如在上面场景中需要知晓 allItems 和 selectedIds。
- 整个计算过程较难复用。

使用 reselect 则可以解决这些问题。同时，它采用缓存方式，提供了更好的计算性能，应用起来也非常简单。使用 reselect 计算过程如图 6-29 所示。

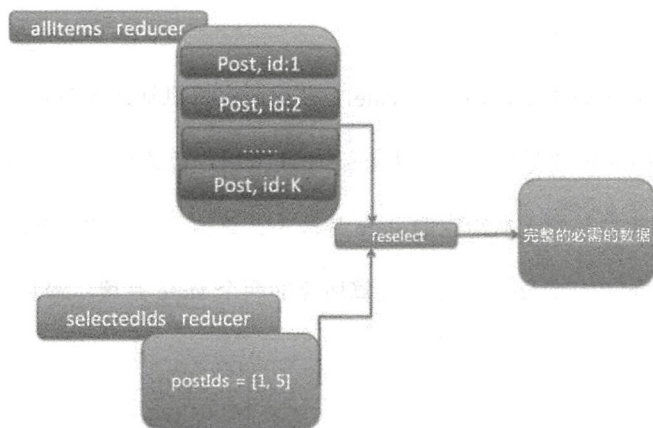


图6-29 使用reselect计算过程示意图

从图 6-29 中可以看出，reselect selector 接手了计算过程，并得到了所需的数据。脱离这个场景，更广义的，应用某个环节的结构将如图 6-30 所示。

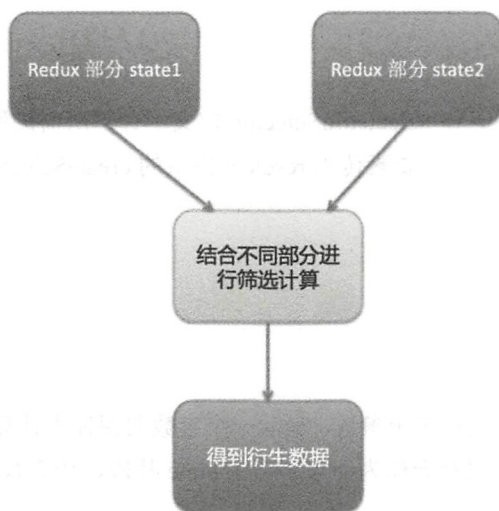


图6-30 更广义的场景示意图

如果我们需要根据 Redux store 的某几部分进行组合计算，才能得到渲染所需的数据，那么这时候就是使用 reselect 的绝佳场景。

回到场景中，看一下具体应用。首先引入依赖：

```
import { createSelector } from 'reselect'
```

然后需要定义以下内容：

- `allItemsSelector` 函数类型，用于从 `state` 中获得所有可供筛选的条目。
- `selectedIdSelector` 函数类型，用于从 `state` 中获得用户已选择的条目 `id`。
- `getItems` 函数类型，用于从给定的 `state` 片段中计算得到所需的数据。

前两项称之为 `inputSelectors`，即定义计算所需的两个 `state` 片段；最后一项为 `resultFunc`，即定义计算规则。

应用代码如下：

```
const allItemsSelector = state => state.allItems;
const selectedIdSelector = state => state.selectedIds;

const getItems = (items, selectedIds) => {
  const selectedItems = items.filter(item => selectedIds.includes(item.id));
  return selectedItems;
}
```

实际上，`allItemsSelector` 和 `selectedIdSelector` 定义了计算所需的所有 `state` 片段，`getItems` 则对应于计算规则。我们把这三个要素传入 `reselect` 提供的 `createSelector` 方法中，便大功告成。

```
export default createSelector(
  allItemsSelector,
  selectedIdSelector,
  getItems
)
```

`createSelector` 函数可以接收多个参数，最后一个参数要保证为计算最终 `state` 片段的规则函数，前面所有参数的计算结果都会作为计算所需的 `state` 片段，传入最后一个参数中进行计算。

```
createSelector(...inputSelectors | [inputSelectors], resultFunc)
```

当 `Redux state` 变化时，`allItemsSelector` 和 `selectedIdSelector` 都会执行，当它们的返回结果发生变化时，`getItems` 将会自动计算出最新的已选择条目。

想象一下，如果 `Redux state` 非常复杂，`allItems` 存在上万个条目，那么计算开销将会大幅度增加。前面提到了 `reselect` 具有缓存特性，实际上，`createSelector` 默认拥有 `size` 为 1 的缓存能力。也就是说，当前后两次 `inputSelectors`（即 `allItemsSelector` 和 `selectedIdSelector`）计算所得结

果没有发生变化时，`resultFunc`（即 `getItems`）将会直接返回缓存结果，因而规避了大量的计算开销。

6.12 Redux store 数据结构扁平化及在 Twitter 中的实践

众所周知，`Redux` 是一个进行数据状态管理的利器，因此不可避免地与数据产生了密切关系。在数据结构设计上，`Redux` 一直强调的数据结构扁平化到底是什么意思？这样设计又有哪些优势呢？本节我们将分析 `Redux` 在数据结构设计上的“最佳实践”。

6.12.1 数据结构扁平化的优化方向和手段

下面我们将通过一个实例场景来分析初级的数据结构设计，并通过一步步优化达到扁平化的目的。

假设有类似于 `Twitter` 或者微博的应用，鼓励用户发布推文，在每一条推文下都会有其他用户的评论。同时每条推文又会隶属一个特定的分类类型，比如用户发布一条关于足球的推文，那么其隶属的分类类型就是“`sport`”；用户发布一条讨论 `JavaScript` 语法细节的推文，那么它就属于“`code`”类型。在这种情况下，基本的但低效的数据结构设计如下：

```
state = {
  activeDataId: '1',
  data: [
    {
      id: '1',
      name: 'code',
      isActive: true,
      tweets: [
        {
          id: '1',
          title: '...',
          description: '...',
          isEditable: false,
          authorId: '...',
        },
        {
          id: '2',
```

```
      title: '...',
      description: '...',
      isEditable: false,
      authorId: '...',
    },
    {
      id: '3',
      title: '...',
      description: '...',
      isEditable: false,
      authorId: '...',
    }
  ],
},
{
  id: '2',
  name: 'sport',
  isActive: false,
  tweets: [
    {
      id: '4',
      title: '...',
      description: '...',
      isEditable: false,
      authorId: '...',
    },
    {
      id: '5',
      title: '...',
      description: '...',
      isEditable: false,
      authorId: '...',
    },
    {
      id: '6',
      title: '...',
      description: '...',
```



```

        isEditable: false,
        authorId: '...',
      },
    ],
  },
],
},
]
}

```

如上所示，我们按照推文的不同类型进行了分类，列举了 id 为 1 的推文对应“code”类型，id 为 2 的推文对应“sport”类型。在每种分类下，都使用 tweets 数组来存储对应分类下所有推文的信息。一条典型的推文信息类似于：

```

{
  id: '6',
  title: '...',
  description: '...',
  isEditable: false,
  authorId: '...',
}

```

直观上，这样的数据结构设计使得数据嵌套层次非常深。如此便会带来一些问题，比如想修改一条推文的属性时，就需要深入多层。当修改“sport”类型下 id 为 4 的推文信息时：

```
state.data[1].tweets[0].description = '';
```

其实这还是完全简化的实现。在真实的开发场景下，我们并不知晓目标数据对应的该路径下数组的 index 值，需要对 data 和 tweets 数组进行遍历筛选出指定的 name 和 id。另外，还需要确保数据访问的安全性，比如 state.data[1].tweets[0].description，如果 state.data 数组中不存在 index 值为 1 的数据，就会报错。这些逻辑可能都会落到 reducer 函数中，使得 reducer 函数变得臃肿且难以维护。此外，在效率性能上，如果数据很大，或者 tweets 数组存储了上万条数据，那么对于目标推文的查找将会极其耗时。

在这样的数据结构中，我们还未加入每条推文对应的评论信息，数据就已经变得非常复杂了。推荐的一种做法是优先使用 object 类型维护数据，使得查找等操作更加方便。事实上，只有需要严格保证数据顺序时，才应该考虑使用数组包裹对象的方式。

因此，对于上述结构的优化设计，首先想到就是使用 object 代替数组。对于先前的 tweets 数组，改造如下：

```
tweets: {
  byId: {
    '1': {
      id: '1',
      title: '...',
      description: '...',
      isEditable: false,
      authorId: '...',
    },
    '2': {
      id: '2',
      title: '...',
      description: '...',
      isEditable: false,
      authorId: '...',
    },
    '3': {
      id: '3',
      title: '...',
      description: '...',
      isEditable: false,
      authorId: '...',
    },
  },
  allIds: ['1', '2', '3']
}
```

这里设计了 `byId` 对象，它的工作是以键值对的形式承载每一条推文信息。同时增设了 `allIds` 字段，以备需要时使用。在这种情况下，将大大提高对特定推文的访问效率。事实上，若一个对象属性的访问复杂度稳定在 $O(1)$ ，其成本将远远高于数组的查找复杂度 $O(n)$ 。

接下来，加入推文评论数组 `comments`。

```
tweets: {
  byId: {
    '1': {
      id: '1',
      title: '...',
```

```

    description: '...',
    isEditable: false,
    authorId: '...',
    comments: ['1', '3', '4', '7']
  },
  '2': {
    id: '2',
    title: '...',
    description: '...',
    isEditable: false,
    authorId: '...',
    comments: ['2', '6']
  },
  '3': {
    id: '3',
    title: '...',
    description: '...',
    isEditable: false,
    authorId: '...',
    comments: ['5', '8', '9', '10']
  },
},
allIds: ['1', '2', '3']
}

```

我们已经吸取经验教训，在上面的数据结构设计中，对于每条推文都使用一个数组来记录评论 id，而不是采用如下深层嵌套的形式。

```

'1': {
  id: '1',
  title: '...',
  description: '...',
  isEditable: false,
  authorId: '...',
  comments: {
    id: '1',
    description: '...',
    isEditable: false,

```

```
      authorId: '...',  
      tweetId: '1',  
    },  
  ],  
},
```

这样将评论信息和推文信息“解耦”，使用评论 id 进行连接，减少了嵌套层次，这就是扁平化的一种体现。同时，对于评论信息数据的设计，同样按照 tweets byId 形式进行。

```
comments: {  
  byId: {  
    '1': {  
      id: '1',  
      description: '...',  
      isEditable: false,  
      authorId: '...',  
      tweetId: '1',  
    },  
    '2': {  
      id: '2',  
      description: '...',  
      isEditable: false,  
      authorId: '...',  
      tweetId: '2',  
    },  
    '3': {  
      id: '1',  
      description: '...',  
      isEditable: false,  
      authorId: '...',  
      tweetId: '1',  
    }, ...  
  },  
  allIds: ['1', '2', '3', '4', '5', ...]  
}
```

这样的扁平化设计，极大地方便了对数据的访问和更新，有利于 reducer 中逻辑的梳理和编写。下面我们把所有数据组合到一起。

```
state = {
  topics: {
    '1': {
      id: '1',
      name: 'code',
      isActive: true,
      tweets: ['1', '2', '3']
    },
    '2': {
      id: '2',
      name: 'sport',
      isActive: false,
      tweets: ['4', '5', '6']
    }
  },
  tweets: {
    byId: {
      '1': {
        id: '1',
        title: '...',
        description: '...',
        isEditable: false,
        authorId: '...',
        comments: ['1', '3', '4', '7']
      },
      '2': {
        id: '2',
        title: '...',
        description: '...',
        isEditable: false,
        authorId: '...',
        comments: ['2', '6']
      },
      '3': {
        id: '3',
        title: '...',
        description: '...',
```

```

        isEditable: false,
        authorId: '...',
        comments: ['5', '8', '9', '10']
      }, ...
    },
    allIds: ['1', '2', '3', ...]
  },
  comments: {
    byId: {
      '1': {
        id: '1',
        description: '...',
        isEditable: false,
        authorId: '...',
        tweetId: '1',
      },
      '2': {
        id: '2',
        description: '...',
        isEditable: false,
        authorId: '...',
        tweetId: '2',
      },
      '3': {
        id: '1',
        description: '...',
        isEditable: false,
        authorId: '...',
        tweetId: '1',
      }, ...
    },
    allIds: ['1', '2', '3', '4', '5', ...]
  }
}

```

结合实践，我们将演示不同的操作方式，以验证这样的数据结构设计的便捷性。

(1) 增加一条推文：只需要在 tweets 对象的 byId 属性中增加相关信息，同时将推文 id 添加到 tweets.allIds 数组中即可。同时 topics tweets 数组也需要更新相应 tweets 数组的 id。

(2) 编辑一条推文: 只需要在 `tweets` 对象的 `byId` 属性中找到对应的推文 `id` 进行更改即可。

(3) 删除一条推文: 只需要在 `tweets` 对象的 `byId` 属性中找到对应的推文 `id` 执行删除操作, 同时在 `tweets.allIds` 数组中删除该推文 `id` 即可。另外, `topics tweets` 数组也要删除相应 `tweets` 数组的 `id`。

(4) 对一条推文增加一条评论信息: 只需要在 `comments` 对象的 `byId` 属性中增加该评论信息数据, 同时在 `comments allIds` 数组中加入评论 `id` 即可。另外, 在对应的推文中, 需要在 `comments` 数组下加入该评论 `id`。

(5) 编辑一条评论信息: 只需要在 `comments` 对象的 `byId` 属性中找到对应评论进行更新即可, 不需要在 `tweets` 数组中进行任何操作。

至此, 读者应该能够体会到扁平化数据结构的优势。综合而言, 它解决了在数据结构设计上信息重复和深层嵌套的问题, 也使得对数据的操作更加简单、高效。在代码层面, 使得 `reducer` 函数的编写更加清晰, 减少了潜在问题的出现, 应用性能也有所提高。

6.12.2 Twitter 在基于 Redux 架构下的数据实践

Twitter 前端经过重构, 已经完全迁移到 `React+Redux+PWA` 技术栈, 了解 Twitter 的 `Redux store` 组织架构, 这对于在复杂场景下进行前端数据学习, 以及 `React`、`Redux` 数据流设计很有意义。

一条推文如图 6-31 所示

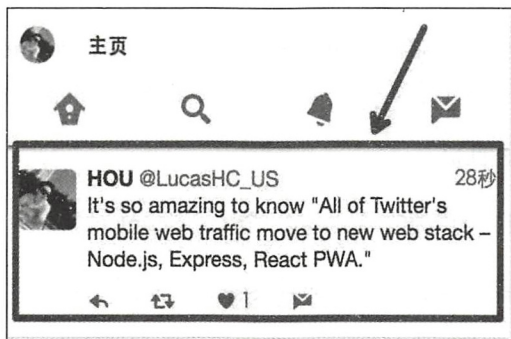


图6-31 一条推文

借助于 `React Developer Tools (RDT)`, 我们可以看到每一时刻的 `store` 快照, 如图 6-32 所示。

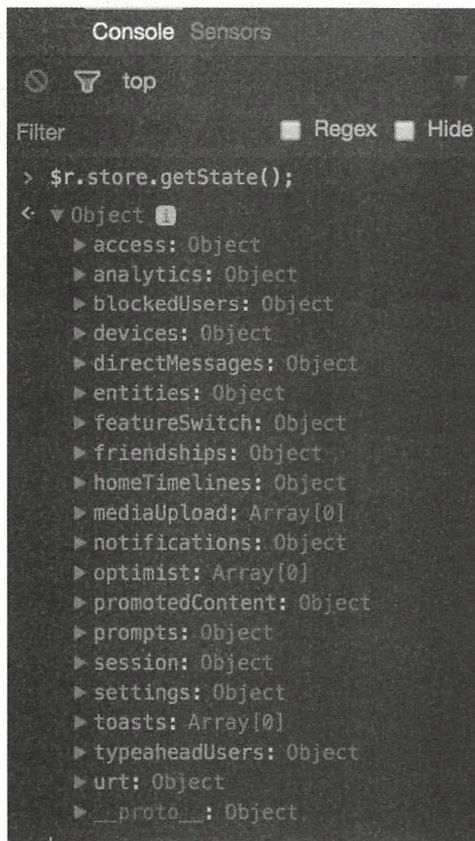


图6-32 数据快照

一条推文的信息数据全部存储在 `entities/tweets/entities` 中，即 `entities/tweets/entities` 中存储了所有推文的信息数据；在这个字段中，每一条推文都是一个键值对类型的 JavaScript 对象：

- `key` 对应于该条推文的 `id`。
- `value` 对应于该条推文的数据，它也是一个 JavaScript 对象。

将第一条推文展开，我们能看到推文的具体信息，如图 6-33 所示。

了解了推文结构后，接下来看一下 Twitter 首页的时间线（timeline）设计。在 Twitter 首页的时间线上，用户可以向下滑动加载更早的推文内容，也可以上拉刷新获取最新的推文信息。直观上，时间线状态一定包含了个人主页所要展示的所有推文信息。通过推文 `id` 和 `entities/tweets/entities` 中的推文相匹配，并最终在时间线上展示，如图 6-34 所示。

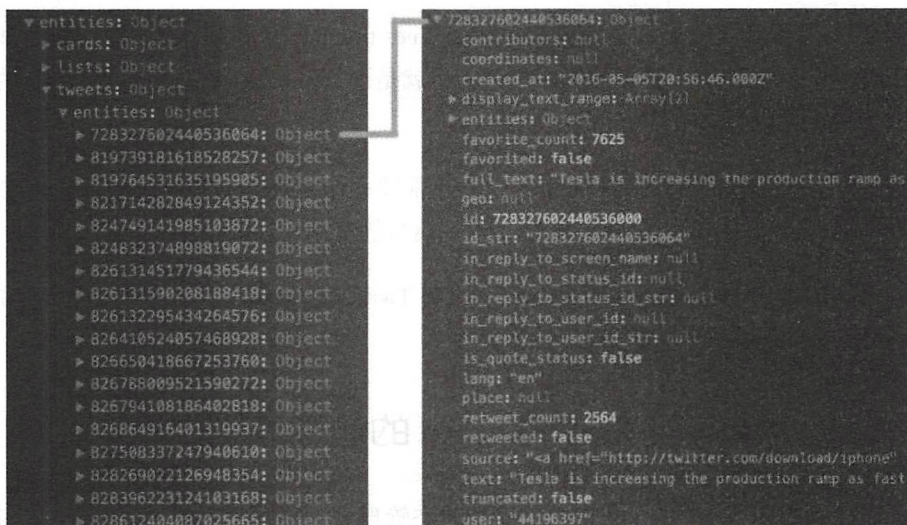


图6-33 推文的具体信息

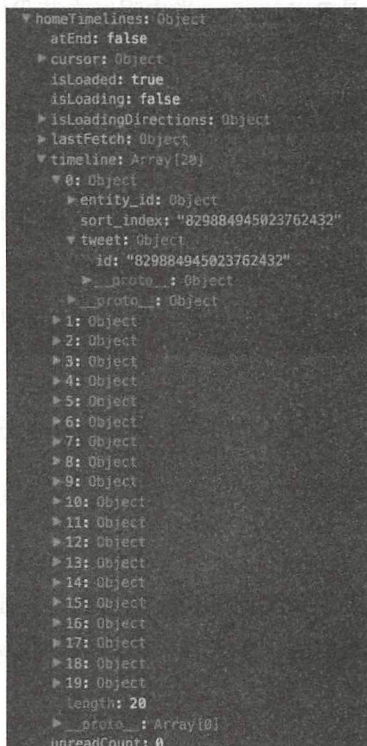


图6-34 在时间线上展示推文信息

每个用户的首页时间线信息都可从 `homeTimelines/timeline` 中找到。首页时间线上展示的推文顺序，则按照 `timeline` 这个数组的顺序排列。也就是说，`timeline` 数组 `index` 值为 0 的条目，对应于在首页时间线上看到的第一条推文。

我们看到，在 `homeTimelines/timeline` 中并不需要存储完整的推文数据，而是通过推文 `id` 和 `entities/tweets/entities` 中的推文进行匹配。这是典型的扁平化数据结构设计的体现。

至此，我们已经了解了扁平化数据结构设计在 Twitter 中的体现。关于其更进一步的设计，本书不再展开，有兴趣的读者可以自行探索。

6.13 React state 和 Redux state 的选取原则

通过学习，我们了解到 Redux 的目标是把状态管理做到极致。实际上，React 本身也兼具状态管理的功能。那么在什么场景下需要使用 Redux 或者 React 进行状态管理呢？应该将数据存储在 React 组件的 `state` 当中还是 Redux store 当中呢？本节我们就从 Redux 和 React 状态管理的差别入手来进行分析。

在 React 中 `state` 被维护在相应组件的内部。当某个 `state` 需要与其他组件共享时，我们可以通过 `props` 来完成组件间通信。从实践上看，需要相对顶层的组件维护共享的 `state` 并提供修改该 `state` 的方法，`state` 本身和修改方法都需要通过 `props` 传递给子孙组件。

在使用 Redux 时，`state` 被维护在 Redux store 当中。任何需要访问并更新 `state` 的组件都需要订阅 Redux store，通常借助于容器组件来完成。由此可见，Redux 对数据采用集中管理的方式。

下面我们试图从数据持久度、数据消费范围这两个方面进行分析。

在数据持久度上，不同状态的数据大体分为三类：快速变更型、中等持续型和长远稳定型数据。

快速变更型数据在应用中代表了某些原子级别的信息，其显著特点是变更频率最快。比如一个文本输入框数据值，可能会随着用户输入在短时间内持续发生变化。这类数据显然更适合维护在 React 组件之内。

对于中等持续型数据，当用户浏览或使用应用时，这类数据往往会在页面刷新前保持稳定。比如从异步请求接口通过 AJAX 方式得来的数据；或者在个人中心页面，用户编辑信息提交的

数据。这类数据较为通用，也会被不同组件所需要，在 Redux store 中维护，并通过 connect 方法进行连接，相比于使用 state 进行维护，这是更好的选择。

长远稳定型数据是指在页面多次刷新或者多次访问期间都保持不变的数据。因为 Redux store 会在每次页面加载后都重新生成，因此这类数据显然应该存储在 Redux 以外的其他地方，比如服务端数据库或者本地存储中。

下面我们从另一个维度：数据消费范围来分析。数据很重要的一个特性体现在消费层面，即需要使用多少组件，我们以此来区分 React 和 Redux 的不同分工。广义上，有越多的组件需要消费同一种数据，在 Redux store 中维护这种数据就越合理；反之，如果某种数据隔离于其他数据，只服务于应用的某一部分，那么由 React 维护更加合理。

具体来看，在 React 中共享数据应该存在于最高层组件中，由此组件进行一层层传递。在 props 传递深度上，如果只需要一两个层级就能满足消费数据的组件需求，那么这样的跨度是可以接受的；反之，如果跨越层级很多，那么关联到的所有中间层组件都需要进行接力赛式的传递，这样显然会增加很多乏味的传递代码，也破坏了中间层组件的复用性。这个时候使用 Redux 维护共享状态，合理设置容器组件，通过 connect 来打通数据，就是一种更好的方式。

一些完全不存在父子关系的组件需要共享数据，比如前面提到的一个页面需要多处展示用户头像的场景，则往往会造成数据辐射分散的问题，这对于 React 模式的状态管理十分不利。在这种场景下，使用 Redux 同样是更好的选择。

如果应用有跟踪状态的功能，比如需要完成“重放”“返回”或者“Redo/Undo”等类似的需求，那么 Redux 无疑是最佳选择。因为 Redux 天生擅长于此：每一个 action 都描述了数据状态的变化和更新，对数据的集中管理非常方便进行记录。

最后，在什么情况下该使用哪种数据管理方式，是使用 React 维护 state 还是使用 Redux 集中管理，这个讨论不会有定论，这需要开发者对 React 和 Redux 有深入理解，并结合场景需求来进行选择。

6.14 本章小结

任何一个类库及其生态的发展，都离不开社区的贡献。我们对 React、Redux 的学习也应当放开眼界，博采众长。在本章中，我们对 React、Redux 的“热点”问题进行了探索，相信读者

通过学习会有更加深刻的理解。同时，本章针对 React、Redux 开发的真实场景中存在的问题和解决方案进行了剖析，其中很多内容来自社区，已经成为所谓的“最佳实践”或通用的解决方案。

这些方案都是通过解决真实问题得来的，解决思路或巧妙或优雅，都是 React、Redux 的思想结晶。学习任何一项技术，拥抱社区，加入讨论，除了贡献自己的知识积累和沉淀，更能对个人的技术水平起到深化和升华的作用。

第 7 章

单页面应用代码分割

随着前端技术的飞速发展，浏览器端脚本在应用中所扮演的角色越来越重要。单页面应用（Single Page Application）模式也被很多站点所采用。简单来说，单页面应用仅在 Web 页面初始化时加载相应的 HTML、JavaScript 和 CSS 资源，一旦页面加载完成，就不会因为用户的操作而重新加载或跳转页面了。这样做的好处显而易见：因为避免了页面间跳转和加载，用户得到了更好的交互体验；在技术层面，服务器压力更小，前后端分离更清晰。

当然，这种模式也不可避免地出现了初次加载耗时过长的情况。在前端工程化、性能优化等话题火热的背景下，“代码分割”这一概念便应运而生。借助于先进的浏览器特性，以及工程化工具的帮助，“代码分割”能有效解决传统的单页面应用脚本过大、加载耗时长的问题，因此备受推崇且得到广泛采用。

这一章，就让我们深入讨论 React 技术中的代码分割话题。

7.1 React 和代码分割

有一些使用 React 技术栈开发的产品都属于单页面应用，因此就会面临优化打包资源的问题。如果打包出单一脚本文件，且此产出文件体积过大，无疑将会增加用户下载的负担，使用户承担不必要的流量和电量消耗，更重要的是会使首屏性能下降，从而影响产品体验。

那么如何为包“瘦身”呢？针对此优化点，近些年出现了很多先进的技术，比如 Tree Shaking、代码分割（Code Splitting）等。Tree Shaking 技术并不难理解，它往往依赖工程化工具分析实现。而代码分割的完成，则需要开发者具有丰富的经验。

7.1.1 代码分割的意义和普遍做法

代码分割，就是将打包后的代码按照某种方式进行分割，即分割成切片，也就是实现不同逻辑模块的脚本文件，再按照相应逻辑对所需切片进行懒加载或按需加载。这样一来，某些从初始母本中被分割出来的切片脚本，也许在用户浏览周期中永远不会被加载。这样做有什么意义呢？

让我们先来回顾一下不采用代码分割的 SPA 方式。

如图 7-1 所示，从生产到上线阶段，所有的应用脚本都被打包成 `bundle.js`。在不采用服务端渲染的情况下，这个 `bundle.js` 包含了页面所有组件的创建、加载等逻辑，也包含了应用脚本的所有依赖，比如必备的 `React`、`react-dom` 等。如果使用了 `Redux` 进行数据状态管理，使用了第三方 UI 库，使用了 `react-router` 包进行路由处理等，那么这个 `bundle.js` 的大小可想而知。

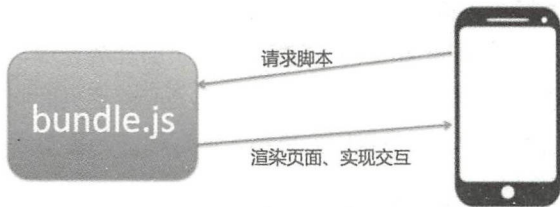


图7-1 单页面应用脚本请求示意图

浏览器请求并加载这个 `bundle.js` 之后，执行相应的逻辑，页面才会展示信息，进而用户方可操作，与页面互动。

让我们思考一下整个过程带来的问题：开发者对应用脚本的任何改动，哪怕只改了一个字符串，也都会重新生成一个全新的 `bundle.js`。作为用户，每次 `bundle.js` 发生变化，也都需要重新下载，下载内容包括一般稳定不变的依赖，如 `React`、`react-dom`、`Redux`、`react-router` 等。

此时，我们会想到“缓存”的概念。缓存的对象应该是一些稳定不易变化的脚本，比如依赖库等。接下来很自然地就会产生代码分割的概念。如图 7-2 所示，我们将 `bundle.js` 分割成 `vendor.js` 和 `app.js`。

- `vendor.js` 打包了所有稳定不易变化的逻辑，往往是应用的依赖库。
- `app.js` 是业务代码，经常会发生迭代变化。

在这种设计下，我们可以对 `vendor.js` 进行缓存——在任何有机会使用缓存的层面进行，比如开发者可以给 `vendor.js` 文件加上 hash 码，设置合适的浏览器端缓存（`max-age`）；也可以根据

业务需要，设置 CDN 缓存等。

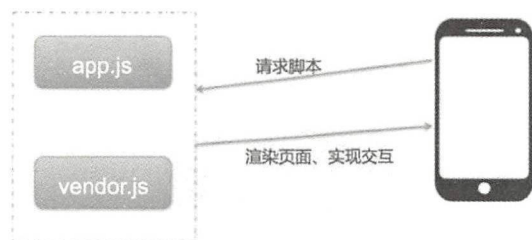


图7-2 脚本拆分示意图

这样的分割收益很明显：在 `vendor.js` 被合理缓存的情况下，无论业务脚本发生任何变化，我们都只加载 `app.js` 即可，这样明显完成了“瘦身”，对于稳定的第三方依赖库，减少了不必要的重复加载。

7.1.2 基于业务的代码分割

有了以上认知，我们进一步思考：对 `app.js` 脚本是否也可以分割？

比如应用包含了首页、登录页和设置页三个页面。未登录用户初次进入页面后，将进入登录页面，只有成功登录之后才能进入首页。在首页中点击“设置”按钮，才会进入设置页面。

如果登录失败，那么就没有加载首页内容的必要了。同样，如果用户在首页中不点击“设置”按钮，那么设置页面的相关内容也不需要一开始就进行加载。所以，我们可以进行更进一步的分割，如图 7-3 所示。

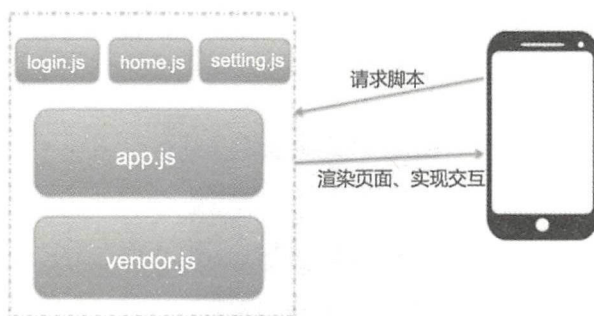


图7-3 进一步分割脚本示意图

由此可知，当进入不同的页面时，所加载的脚本也不尽相同。

在登录页时，只关心登录页脚本、vendor.js 和 app.js，如图 7-4 所示。



图7-4 登录页脚本分割示意图

在首页时，只关心首页脚本、vendor.js 和 app.js，如图 7-5 所示。



图7-5 首页脚本分割示意图

在设置页时，只关心设置页脚本、vendor.js 和 app.js，如图 7-6 所示。

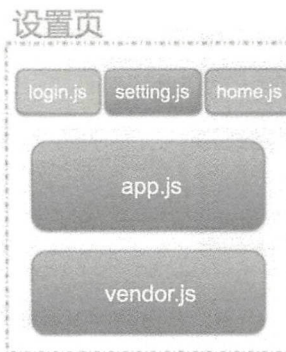


图7-6 设置页脚本分割示意图

7.1.3 合理选择分割维度

在上面的案例中，我们进行了合理的代码分割。但是在实际的开发中，每个应用所面临的需求不同、设计不同，那么如何进行合理的分割设计呢？也就是说，是否存在一些准则或规律，可以帮助开发者进行合理的分割，设置先加载哪些部分，然后再按需加载其他部分呢？

一般来讲，有如下三种分割代码的方式。

- 按照业务逻辑和依赖库分割。
- 按照路由分割。
- 按照组件分割。

按照业务逻辑和依赖库分割，即将 `bundle.js` 分割为 `app.js` 和 `vendor.js`，前面已经做了介绍。接下来了解一下按照路由分割，这种分割方式需要整个应用路由的不同页面彼此较为独立，同时应用中的路由需要对按需加载“有所感知”，以便在合适的时机进行加载。在著名的 `react-router` 库中，就有对实现路由层面代码分割的支持。

需要注意的是，这种分割方式可能会带来冗余。事实上，任何事情都“过犹不及”，设想一下，我们按照 20 个路由页面，分割出 20 个相互独立的脚本，那么这 20 个脚本之间就很容易产生很多重复的代码。

为了弥补这种分割方式带来的弊端，我们也可以按照组件维度进行分割，别忘了 `React` 带来的全新理念之一就是组件化。组件分为“UI 组件”和“容器组件”，在容器组件中，我们可以进行数据请求、数据分发、逻辑处理等；UI 组件只负责接收并展示数据。在基于组件维度的代码分割中，通常的做法是由容器组件控制，在容器组件内实现按需加载 UI 组件，即根据条件判断是否需要加载合适的 UI 组件。

7.1.4 合理选择加载时机

既然涉及按需加载，那么就必然引出一个加载时机的问题。既然是“在需要的时候加载”，那么应该如何定义“需要的时候”呢？

基于此，按照加载时机的不同，分为消极加载（`Passive Preloading`）和积极加载（`Active Preloading`）。

消极加载是指不需要用户额外的交互便进行加载，加载往往由组件的某个生命周期函数或者相关组件的某些行为触发。其核心特征是触发时机不依赖用户的动作。

下面我们使用 `react-loadable` 库来演示消极加载方式。

以一个新闻列表页为例，假设该页面中的每一项都是某条新闻的标题和详情页链接，点击列表中的某一项，便会跳转到该条新闻的详情页，如图 7-7 所示。



图7-7 页面跳转示意图

首先对新闻详情页进行 `Loadable` 化处理，实际上就是按需加载包装。

```
import Loadable from 'react-loadable';

const MyLoadingSpinner = () => {
  <div> Loading... </div>
}

const LoadableNewsPage = Loadable({
  loader: () => import('./NewsDetailPage');
  LoadingComponent: MyLoadingSpinner
})
```

`Loadable` 是一个高阶组件，返回 `LoadableNewsPage` 组件，就完成了动态加载。

在新闻列表页组件 `NewsList` 中，即可按需进行加载。

```
import React, { Component } from 'react';
import { NewsLink } from '../components';

class NewsList extends Component {
  componentDidMount() {
    LoadableNewsPage.preload();
  }
  render() {
    const { newsList } = this.props;
    return (
      <div>
        {newsList.map(
          props => <NewsLink {...props}>
        )}
      </div>
    )
  }
}
```

注意：在新闻列表页组件 `NewsList` 的生命周期函数 `componentDidMount` 中，完成了对新闻详情页的加载。很明显，这样的策略说明新闻列表页和新闻详情页强相关，当新闻列表页被渲染加载时，就表示是时候加载新闻详情页了。

相对地，积极加载则依赖用户的动作和页面进行交互。在上面的场景中，我们捕捉新闻列表中用户鼠标的 `mouseenter` 事件，当鼠标指针悬浮于当前新闻条目上时，才加载这条新闻的详情页。

对应的代码逻辑是当链接标签的 `mouseenter` 事件被触发时，才调用 `LoadableNewsPage.preload`。

```
const NewsLink = () => {
  <div>
    <h1>{props.title}</h1>
    <Link
      to = {`/news/${props.id}`}>
```

```
    onMouseEnter={ LoadableNewsPage.preload }  
  >  
    Go to Detail  
  </Link>  
</div>  
}
```

由此可见,消极加载和积极加载有不同的加载时机和策略,开发者可以根据需求合理选择。

7.1.5 按需加载实现原理

熟悉按需加载以及 Webpack 2 的读者可能听说过 `require.ensure` 方法,甚至使用过 `require.ensure` 方法来实现按需加载。这个 API 的设计就是用来实现代码分割的,它会单独打包指定的文件,不和主文件打包在一起。但是在本书中,我们不会对 `require.ensure` 方法的使用进行介绍,因为现在已经有了更加规范、更被推荐的方式——动态导入。

在 Webpack 2 中已经实现动态导入,我们需要配置 `syntax-dynamic-import` 这样一个 Babel 插件,便可直接使用。也就是说,目前 Webpack 2 支持动态导入和 `require.ensure` 两种方式实现按需加载,这里主要介绍通过动态导入来实现按需加载。

在默认情况下,导入模块是静态的,这就要求在文件的顶层定义模块的依赖导入。事实上,这样的设计对于 JavaScript 引擎优化非常有意义,但是对于模块按需加载却设置了天然屏障。

在新的动态导入提议中,允许动态导入模块。其用法非常简单。假设 `index.js` 存在两个依赖脚本文件: `a.js` 和 `b.js`。其中,在 `a.js` 中有:

```
const a = () => console.log('I'am a');  
export default a;
```

在 `b.js` 中有:

```
const b = () => console.log('I'am b');  
export default b;
```

在 `index.js` 中可以先加载 `a` 模块,然后按需加载 `b` 模块。

```
import a from './a';  
a(); // I'am a;  
  
const bPromise = import('./b.js');
```

```
bPromise.then(()=>{
  console.log('b.js is loaded dynamically');
});
```

在上面的代码中，关键的一步是在导入 `b` 的过程中会返回一个 `Promise` 对象。这也是实现 `React` 按需加载的理论基础。

接下来，我们看看 `React` 如何与动态导入相结合。对按需加载的关注点在控制加载上，比如如何加载脚本，脚本是否已经加载，如果还没有加载该怎么办，这些都涉及状态以及状态的切换，而管理状态正是 `React` 的特长。

为此，我们抽象实现一个 `Async` 组件，用于按需加载。

```
export default class Async extends React.Component {
  componentWillMount = () => {
    this.cancelUpdate = false;
    this.props.load.then((c) => {
      this.C = c;
      if (!this.cancelUpdate) {
        this.forceUpdate();
      }
    })
  }

  componentWillUnmount = () => {
    this.cancelUpdate = true;
  }

  render = () => {
    const {componentProps} = this.props;
    return this.C
      ? this.C.default
        ? <this.C.default {...componentProps} />
        : <this.C {...componentProps} />
      : null;
  }
}
```

在应用该组件时，需要传入一个名为 `load` 的 `prop`，该 `prop` 是 `Promise` 类型，用于动态导入

其他组件。当 Async 组件生命周期函数执行到 `componentWillMount` 时，实现动态导入所需的组件，并存储为 `this.C`。同时在 Async 的 `render` 方法中判断 `this.C` 和 `this.C.default` 的可用性，进行渲染。

具体应用如下：

```
const MyAsyncComponent = () => <Async load={import('./MyComponent')} />
```

如果需要保留组件 `./MyComponent` 原有的 `props`，则可以设置 `componentProps` 这个 `prop` 进行传递。

```
const MyAsyncComponent = (props) => <Async load={import('./MyComponent')} componentProps={props} />
```

当然，这样的设计并不唯一，更不绝对。还记得前面介绍的那个新闻列表页和新闻详情页使用 `Loadable` 的场景吗？

```
const LoadableNewsPage = Loadable({
  loader: () => import('./NewsDetailPage');
  LoadingComponent: MyLoadingSpinner
})
```

事实上，如果翻看 `react-loadable` 源码，就会发现实现其思路与 `Async` 是相同的，只不过 `react-loadable` 做了更多的个性化和默认处理。

7.2 Redux reducer 层面代码分割

在 `Redux reducer` 层面实现代码分割，目的是对应用进行按需加载。这种按需不仅仅是组件层面的，在使用 `Redux` 进行数据状态管理时，`reducer` 函数往往会随着应用的复杂度上升而变得庞大，此时按需动态导入 `reducer` 是十分必要的。学习以下内容需要读者先行了解 `Redux` 中间件、`Redux combineReducers` 等知识。

下面我们以一个最基本的计数器为例，实现动态导入 `reducer`。假设页面上有两个计数器：`A` 和 `B`。

```
import {createStore, combineReducers} from 'redux';

const rootReducerInitialState = {
  rootA: 1,
```

```

    rootB: 1
  }
const rootReducer = (state = rootReducerInitialState, action) => {
  switch (action.type){
    case 'INC_A': return {...state, rootA: state.rootA + 1};
    case 'DEC_A': return {...state, rootA: state.rootA - 1};
    case 'INC_B': return {...state, rootB: state.rootB + 1};
    case 'DEC_B': return {...state, rootB: state.rootB - 1};
  }
  return state;
}

const store = createStore(combineReducers({app: rootReducer}));

```

我们设置'INC_A'、'DEC_B'、'INC_B'、'DEC_B'来分别表示计数器 A 和 B 的加减 action。

使用 `combineReducers` 只是在代码组织层面上进行分割，并不是真正意义上的按需加载。在实际开发中，我们更应该关注的是如何实现动态导入或动态替换。

Redux store 已经有了一个现成的方法用于替换 reducer，即 `store.replaceReducer`，借助于它，我们可以动态地实现传递一个新的 reducer 来取代旧的 reducer。先假设在上述代码的基础上，已经有了初始状态的 reducer，此时如果需要再添加一个 `splitReducerA`，则可以这样做：

```

const codeSplitA = {
  aValue: ''
}

const splitReducerA = (state = codeSplitA, action) => {
  switch (action.type){
    case 'set_A': return {...state, aValue: action.value};
  }
  return state;
}

store.replaceReducer(combineReducers({
  app: rootReducer,

```

```
    aModule: splitReducerA
  }));
```

当需求扩充，又需要一个 `splitReducerB` 时，则可以这样做：

```
const codeSplitB = {
  bValue: ''
}

const splitReducerB = (state = codeSplitB, action) => {
  switch (action.type){
    case 'set_B': return {...state, bValue: action.value};
  }
  return state;
}

store.replaceReducer(combineReducers({
  app: rootReducer,
  aModule: splitReducerA,
  bModule: splitReducerB
}));
```

如此，我们使用 `store.replaceReducer` 就完成了一个又一个新 reducer 的动态导入。

但是这样做还存在一个问题，就是每次在调用 `store.replaceReducer` 时，都需要将旧的 reducer 一道包裹进来。为此可以进行优化，方案是维护一个正处于运行状态的 reducer 列表，同时设计 `getNewReducer` 方法，将新导入的 reducer 和现有的 reducer 列表合并，然后再进行 `store.replaceReducer` 替换。

```
const asyncReducers = {};

const getNewReducer = newModuleInfo => {
  asyncReducers[newModuleInfo.name] = newModuleInfo.reducer;

  store.replaceReducer(combineReducers({
    app: rootReducer,
    ...asyncReducers
  }));
}
```


此时，如果需要再新导入 reducer C，就可以这样做：

```
const codeSplitC = {
  CValue: ''
}

const splitReducerC = (state = codeSplitC, action) => {
  switch (action.type){
    case 'set_C': return {...state, cValue: action.value};
  }
  return state;
}

getNewReducer({name: 'cModule', reducer: splitReducerC});
```

上面所提到的问题，还在于在使用 createStore 创建 store 的情况下，createStore 方法可以接收第二个参数来表示应用的初始状态。这样的用法在很多场景下都很有用处，比如可以保证用户跳转后重新返回页面时仍然保持离开前的状态等。示例代码如下：

```
const storeInitialState = {
  app: { rootA: 10, rootB: 12 },
  aModule: { aValue: 'saved A value' },
  bModule: { bValue: 'saved B value' },
}

const store = createStore(combineReducers({app: rootReducer}), storeInitialState);
```

上面这段代码应用了 storeInitialState 作为初始状态。但是这样做，我们很快就得到了错误提示：

```
Unexpected keys "aModule", "bModule" found in preloadedState argument passed to createStore.
Expected to find one of the known reducer keys instead: "app". Unexpected keys will be
ignored.
```

问题在于初始状态是完整的，它已经包含了后续需要注入的 reducer 所对应的初始状态，即 aModule、bModule。而当应用启动时，后续 reducer 并没有被载入，它需要被动态导入，这样一来，在初始创建 store 时，便缺少这些 reducer 来匹配完整的初始状态。

对于上述问题，一种常见的解决办法是侦测出所缺少的（即后续需要导入的）reducer，然后手动加入默认匹配数据状态的伪 reducer。

```
const combineLazyReducers = (reducers, initialState) => {
  let reducerKeys = new Set(Object.keys(reducers));
  Object.keys(initialState)
    .filter(k => !reducerKeys.has(k))
    .forEach(k => {
      reducers[k] = state => state === undefined ? null : state
    });

  return combineReducers(reducers);
}
```

在上面的代码中遍历了 `initialState`，对于匹配不上的 `reducer`，则自动匹配 `null`。

```
state => state === undefined ? null : state
```

另一种解决办法是使用代理（Proxy），代理方式允许对一个对象的访问进行拦截，因此我们可以对 `reducer` 进行代理。代码如下：

```
const combineLazyReducers = (reducers, initialState) => {
  initialState = initialState || {};
  let handler = {
    ownKeys(target) {
      return Array.from(new Set([...Reflect.ownKeys(target), ...Reflect.ownKeys(
initialState)]));
    },
    get(target, key) {
      return target[key] || (state => state === undefined ? null : state);
    },
    getOwnPropertyDescriptor(target, key) {
      return Reflect.getOwnPropertyDescriptor(target, key) || Reflect.getOwn
PropertyDescriptor(initialState, key);
    }
  };

  return combineReducers(new Proxy(reducers, handler));
}
```

Proxy 可以被认为是一个构造函数，它可以接收两个参数：目标对象（target）和句柄对象（handler）。返回一个代理对象 Proxy，主要用于从外部控制对对象内部的访问。

关于这种处理方式本书不再展开介绍，思路大同小异。上述两种方式都可完全满足需求。

本节我们从另一个角度分析了 Redux reducer 代码分割、按需加载的可行性。其主要实现思路是对 `store.replaceReducer` 的应用，这对于理解 Redux 的核心思想、开发出更加复杂的应用具有重要意义。

7.3 代码分割工程实例

经过上一节的理论知识储备，本节我们将通过一个工程实例来分析代码分割技术。这个工程实例采用了 Didier FRANC 的线上项目，我们对此项目进行剖析，并把它整理到本书的代码示例项目中，感兴趣的读者可以进行了解。

作为典型的单页面应用，其包含了三个主要页面：登录页、注册页和首页。

登录页面：访问地址/login，如图 7-8 所示。

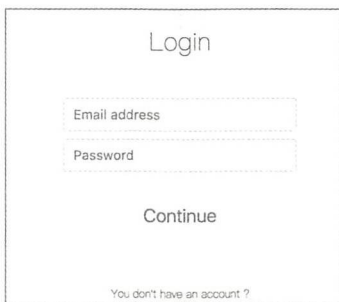
A screenshot of a login page titled "Login". It features two input fields: "Email address" and "Password", both with dashed borders. Below the fields is a "Continue" button. At the bottom, there is a link that says "You don't have an account ?".

图7-8 登录页面

注册页面：访问地址/sign，如图 7-9 所示。

A screenshot of a sign up page titled "Sign up". It features two input fields: "Email address" and "Password", both with dashed borders. Below the fields is a "Continue" button. At the bottom, there is a link that says "Already have an account ?". The page header includes the text "redux-react-starter" and "★ Star 130".

图7-9 注册页面

首页：访问地址/，如图 7-10 所示。

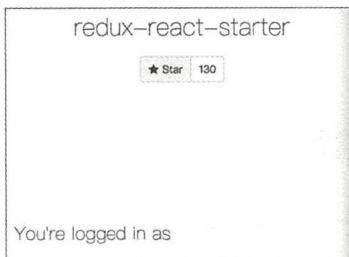


图7-10 首页

7.3.1 依赖和业务分离

通过前文描述，我们已经理解了将整个应用脚本分割为业务脚本和依赖库脚本的意义。一般项目中都会采用 Webpack 来完成这项工作，这里会使用到 CommonsChunkPlugin 插件。

在 webpack.config.prod.js 文件中，配置：

```
module.exports = {
  entry: {
    main: resolve(__dirname, '../src'),
    vendor: [
      'react',
      'react-dom',
      'react-redux',
      'react-router-dom',
      'redux',
      'redux-thunk',
      'emotion',
    ],
  },
  // ...
  plugins: [
    // ...
    new webpack.optimize.CommonsChunkPlugin({
      names: ['vendor', 'manifest'],
```

```

    }},
    // ...
  ],
}

```

entry.vendor 定义了需要打包在一起的依赖库数组。在此实例中，分别是 react, react-dom, react-redux, react-router-dom, redux, redux-thunk, emotion，将来它们会被打包成 vendor.js。

在 plugins 中，我们使用 CommonsChunkPlugin 进行代码分割。

具体的业务代码相对简单，这里不再展开介绍，感兴趣的读者可以自行到线上代码仓库中进行了解。

运行应用后，脚本资源显示如图 7-11 所示。

Asset	Size	Chunks	Chunk Names
0.7748a13f9ec424c70957.js	443 bytes	0 [emitted]	
main.9251a5ace083e813caf8.js	45.7 kB	1 [emitted]	main
vendor.b52d26598622493ad85d.js	191 kB	2 [emitted]	vendor
manifest.c3411e64e069612133d6.js	1.44 kB	3 [emitted]	manifest
index.html	1.64 kB	[emitted]	
sw.js	8.75 kB	[emitted]	
[4] ./node_modules/react-router-dom/es/index.js + 30 modules	77.6 kB	[2] [built]	
[8] ./node_modules/react-redux/es/index.js + 14 modules	38.3 kB	[2] [built]	

图7-11 脚本资源

整个脚本已经被分割为 vendor.js (930KB) 和 bundle.js (1.01MB) 两个文件，在整个应用不包含静态资源如图片、字体的情况下已经将近 2MB。

接下来，我们看看还有哪些设计优化的余地。

7.3.2 按需加载组件

目前，所有的组件都被打包在一起，造成了 bundle.js 的大小超过 1MB，这是难以忍受的。打开 components/app.js 文件：

```

import Login from './Login'
import Signup from './Signup'
import Header from './Header'
import Home from './Home'

const App = ({ user }) => (

```

```
<Body>
  <Header />
  {user.loggedIn ? <Route path="/" component={Home} /> : <Redirect to="/login" />}
  <Route path="/signup" component={Signup} />
  <Route path="/login" component={Login} />
</Body>
)
```

无论用户登录与否，Home 组件都必定会被打包进来。然而，如同之前所讲的，如果一个用户连登录验证都无法完成，则是没有必要加载首页 Home 组件的。为此，我们可以将 Home 组件设计为按需加载，采用动态导入来完成。

```
import Login from './Login'
import Signup from './Signup'
import Header from './Header'

class Home extends React.Component {
  componentWillMount = () => {
    import('./Home').then(Component => {
      this.Component = Component
      this.forceUpdate()
    })
  }
  render = () => (
    this.Component ? <this.Component.default /> : null
  )
}

const App = ({ user }) => (
  <Body>
    <Header />
    {user.loggedIn ? <Route path="/" component={Home} /> : <Redirect to="/login" />}
    <Route path="/signup" component={Signup} />
    <Route path="/login" component={Login} />
  </Body>
)
```


这样就避免了在 app.js 文件一开头就导入 Home 组件,而是只有当用户登录,即 user.loggedIn 为 true 时,才实例化 Home 组件,并在其 componentWillMount 生命周期函数中动态导入真正的 /Home 模块。

当然,也可以使用 Async 组件完成复用。

```
const Home = () => <Async load={import('./Home')} />
```

最终的脚本资源显示如图 7-12 所示。

Asset	Size	Chunks	Chunk Names
0.1c07b12f54d3b88823ec.js	549 bytes	0, 3 [emitted]	
vendor.7486a077152239382a8c.js	273 kB	1, 3 [emitted]	[big] vendor
main.c2af7cefd16dfa355ebb.js	24.8 kB	2, 3 [emitted]	main
manifest.b7b04d031b654f8da44a.js	1.46 kB	3 [emitted]	manifest
index.html	1.37 kB	[emitted]	
sw.js	7.73 kB	[emitted]	

图7-12 脚本资源（最终结果）

其中第一个 0.[chunkhash].js 文件（549 字节）就是老的 Home 组件脚本,我们已经成功进行了分割。

7.3.3 收益与总结

现在,只有当用户登录时,才按需加载 Home 组件。请参考如图 7-13 所示的资源加载瀑布流。

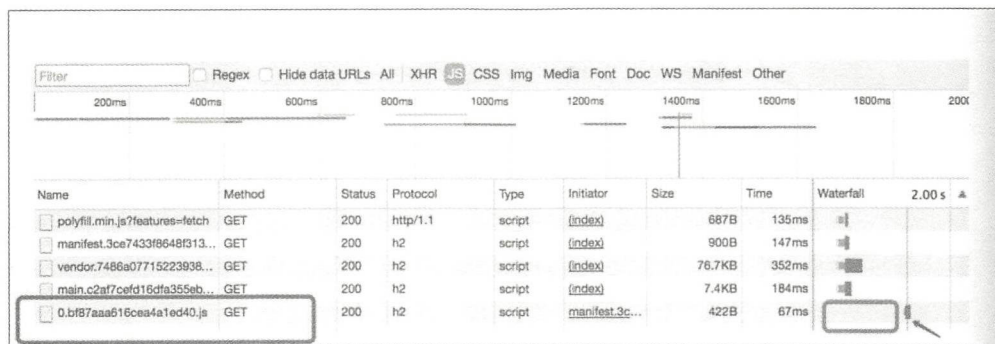


图7-13 资源加载瀑布流

通过代码分割,使性能得到很大提升,这就是代码分割的典型收益。当然,在实际开发中,是否有必要使用此项技术,以及如何使用此项技术,都需要开发者根据现实情况进行考虑。

7.4 本章小结

代码分割不仅仅关系到性能优化,更是一种技术工程设计的体现,它直接影响到用户体验。本章围绕这个主题,对 React 下的代码分割技术进行了梳理与总结,并通过一个单页面应用进行了演示。

第 8 章

React 应用性能优化

通过前面学习，我们了解到 React 不仅仅是一个 UI 库，其背后思想才是引发前端变革的关键。说到前端开发，似乎永远离不开性能这个话题。React 通过虚拟的 DOM 等一系列为开发者津津乐道的手段，保障了性能的高效。可是，它真的是保证开发性能的利器，无往不利吗？在社区中针对 React 应用性能的尝试和实施，似乎也从来没有停止过。

实际上，本书第 6 章所介绍的如 selector 应用、Redux connect 的使用等内容已经涉及性能方面的实践；第 7 章介绍的代码分割更是直接关系到性能的表现。

这一章，我们就从基础和代码细节出发，谈一谈 React 的性能。

8.1 React 应用性能的秘密

说起 React 应用的性能话题，很多读者可能会想到“不要过早地做优化”这条原则。实际上，很多 React 应用的复杂度并不会对性能和产品体验构成挑战。毕竟，高效的虚拟 DOM diff 算法和先进的 React 内部引擎已经做得较为完美了，一般项目需求对性能的压力并不大。

但是对于一些极其复杂的需求，性能优化是无法回避的。如果你开发的是图形处理应用、DNA 检测实验应用、富文本编辑器或者功能丰富的表单型应用，则很容易触碰到性能瓶颈。同样，作为 React 开发者，也需要对性能优化有所了解，这对于理解 React 也是有很大的帮助的。

8.1.1 性能到底指什么

前端开发，自然离不开浏览器，而性能优化大都在和浏览器打交道。我们知道，页面每一

帧的变化都是由浏览器绘制出来的，并且这个绘制频率受限于显示器的刷新频率，所以一个重要的性能数据指标是每秒 60 帧的绘制频率（对应于显示器的 60Hz）。这样进行简单的换算之后，每一帧只有 16.6ms 的绘制时间。

如果一个应用对用户的交互响应处理过慢，需要花费很长的时间来计算更新数据，则会给用户带来极差的使用体验。对于 React 来说，开发者不需要额外关注 DOM 层面的操作。因为 React 通过维护虚拟的 DOM 及其高效的 diff 算法，可以决策出每次更新的最小化 DOM Batch 操作。我们编写的 React 组件状态都处于 JavaScript 层面，虚拟的 DOM 全部由 JavaScript 对象表示。因此，React 最大限度地避免了开发者对 DOM 的直接操作。

一个典型的性能问题是：在不合适的时间进行了 DOM 操作，从而引发强制刷新或者说布局震荡。具体可以通过以下代码来理解。

```
// 读
var l1 = someNode1.style.left;

// 写（无效布局）
someNode1.style.left = (l1 * 2) + 'px';

// 读（触发布局）
var l2 = someNode2.style.left;

// 写（无效布局）
someNode2.style.left = (l2 * 2) + 'px';
```

上述代码存在的问题是没有将读取和写入功能分开，这会强制浏览器重新计算布局。因为在写入样式时，浏览器并不会马上进行布局。但是在第二行代码执行完毕之后，第三行代码需要读取元素 left 值，因此浏览器需要在读取之前触发布局，做出更新，以得到 l2 的准确信息。

但是在 React 世界里，可以通过声明式来完成。

```
<SomeComponent style={{left: this.state.left}} />
```

为了完成上述操作逻辑，只需要简单地进行 state 更新即可。

```
this.setState({left: this.state.left * 2})
```

这就是一个很简单的 React 规避 DOM 操作的例子。把与上述相同或类似的优化过程，交给 React 来处理完成。当然，这样的简单优化同样可以用非 React 技术来实现。

实际上,使用 React 能完成的性能优化,使用纯原生的 JavaScript 都能做到,甚至做得更好。但是经过 React 统一处理后,大大节省了开发成本,同时也降低了应用性能对开发者优化技能的依赖。

这仅仅是一个强制同步更新的例子,目的是让读者了解 React 开发性能的真实面目。

8.1.2 正确理解 React 虚拟的 DOM 带来的优化

图 8-1 大体展示了浏览器渲染和操作 DOM 的过程。

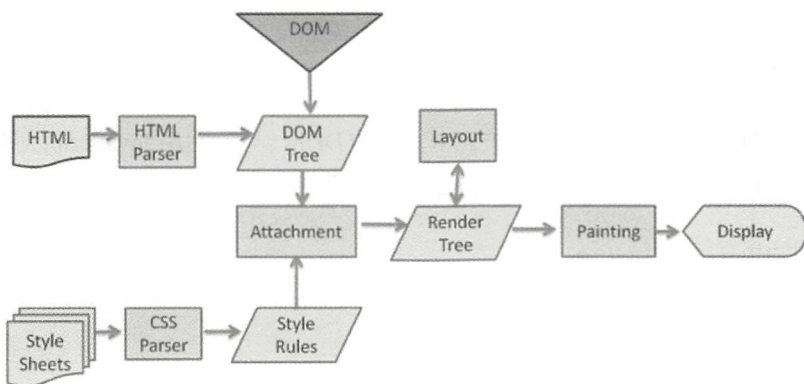


图8-1 浏览器渲染和操作DOM的过程

在真实的 DOM 中,浏览器解析 HTML 之后,渲染引擎负责展现和渲染页面样式。还需要配合解析 CSS,构建渲染树(Render Tree)。这个过程对应于图中的 Attachment。

当页面首次创建完成之后,布局(Layout)过程给出了页面需要做的更新,这同样会触发渲染树的重新构建和绘制(Painting)。

当进行 DOM 操作时,代码如下:

```
document.getElementById('elementId').innerHTML = "New Value"
```

从原理层面讲,这一过程包含以下几个阶段。

- (1) 浏览器解析 HTML 片段。
- (2) 删除之前 id 为 'elementId' 的子元素。
- (3) 使用新的 "New Value" 更新 DOM。

(4) 为相关节点的父节点和子节点重新计算 CSS 样式。

(5) 在显示屏上完成新的布局或绘制。

(6) 在渲染树中进行新的样式绘制。

在上述过程中，重新计算 CSS 样式和更新布局涉及比较复杂的算法和通信过程。实际上，这些操作 DOM 的方法都是浏览器暴露出来的 API，对性能的影响相对较大，并且每一次更新都可能重复触发上述过程。可想而知，代价是非常昂贵的。

为了规避滥操作 DOM 而引发性能隐患，React 不建议开发者直接更新真实的 DOM 节点，而是维护一套虚拟的 DOM，通过使用 diff 算法等手段触发更新（需要注意：设计虚拟的 DOM 不全是为了提升性能）。这种做法能够最大限度地保证 React 应用的性能。

其实虚拟的 DOM 就是在内存中维护一个真实的 DOM 结构。它由 JavaScript 对象描述，React 主要通过以下几种方式来保证虚拟的 DOM diff 算法和更新的高效性能。

- 高效的 diff 算法。
- Batch 操作。
- 摒弃脏检测更新方式。

当任何一个组件使用 `setState` 方法时，React 都会认为该组件变“脏”了，触发组件本身重新渲染。同时因其始终维护两套虚拟的 DOM，其中一套是更新后的虚拟的 DOM；另一套是前一个状态的虚拟的 DOM。通过对这两套虚拟的 DOM 运用 diff 算法，找到需要变化的最小单元集，然后把这个最小单元集应用在真实的 DOM 当中。

而通过 diff 算法找到这个最小单元集，React 采用启发式的思路进行了一些假设，将两棵 DOM 树之间的 diff 复杂度由 $O(n^3)$ 缩减到 $O(n)$ 。

当然，最后还是对真实的 DOM 进行操作的。所以，开发者在应对各种需求时，如果总是能保证对真实的 DOM 进行操作则是最简单的，变动最小，性能最优，就不需要使用 React 本身的 diff 算法了，因为这样的操作在性能上永远领先任何框架。但是这需要开发者具有丰富的经验，并且会耗费极大的设计、开发和维护成本。

说到这里，你一定很想知道 React 的那些大胆假设吧。

- DOM 节点跨层级移动忽略不计。

- 拥有相同类的两个组件生成相似的树形结构，拥有不同类的两个组件生成不同的树形结构。

根据这些假设，React 采取的策略如下：

- React 对组件树进行分层比较，两棵树只会对同一层级的节点进行比较。
- 当对同一层级的节点进行比较时，对于不同的组件类型，直接将整个组件替换为新类型组件。

对于图 8-2 所示的组件结构，我们可以想象：如果子组件 B 和 H 的状态同时发生变化，当遍历到 B 组件时，默认就会更新子组件 H，减少了不必要的消耗。

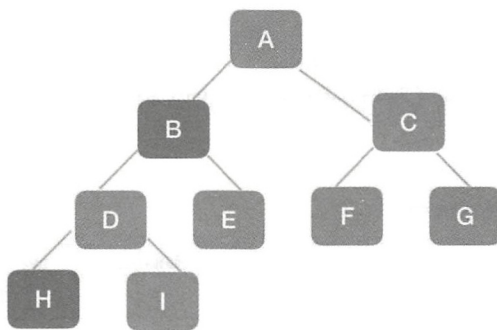


图8-2 组件结构示意图

- 当对同一层级的节点进行比较时，对于相同的组件类型，如果组件的 `state` 或 `props` 发生变化，则直接重新渲染组件本身。开发者可以尝试使用 `shouldComponentUpdate` 生命周期函数来规避不必要的渲染。
- 当对同一层级的节点进行比较时，开发者可以使用 `key` 属性来“声明”同一层级节点的更新方式。

`reconciliation` 过程决定了真实 DOM 的哪一部分需要更新，即前文提到的最小单元集。`setState` 方法引发了“蝴蝶效应”，并通过创新的 `diff` 算法找到需要更新的最小单元集，但是更新并不一定同步进行。实际上，React 会进行 `setState` 的 Batch 操作，通俗地讲，就是“积攒归并”一批变化后，再统一进行更新。显然这是出于对性能的考虑。

8.1.3 使用 React.addons.Perf 进行性能调试

工欲善其事，必先利其器。在正式介绍代码设计层面的性能优化之前，我们有必要先了解一下性能调试神器——React.addons.Perf。在 React 0.13 及以下版本中，可以在 react/addons 文件中找到 React.addons.Perf 的定义；在 React 0.14 及以上版本中，react-addons-perf 已经独立，需要开发者自行配置安装。但是很遗憾，在 React 0.16 版本中，React 将不再支持此插件工具。不过，这不妨碍我们在书中使用它，给读者展现更加直观和清晰的量化数据。

在开发环境下，我们可以得到 React.addons.Perf 对象。在浏览器控制面板中，只需要简单地调用 Perf.start()，然后对应用进行正常的操作，操作结束后，再调用 Perf.stop()，得到的结果如图 8-3 所示。

(index)	Owner > component	Wasted time (ms)	Instances
0	"Todos > TodoItem"	102.76999999977124	1000

Total time: 132.71 ms

图8-3 React.addons.Perf操作示意图

接下来，可以调用 React.addons.Perf 对象中的其他 API 进行了解

- Perf.printWasted(): 打印可以优化的时间，即我们花费了多少时间在虚拟 DOM 树的创建和 diff 上，但是最终并没有对真实的 DOM 节点进行更新或改动。
- Perf.printInclusive(): 打印花费的总时间。
- Perf.printExclusive(): 打印 Exclusive 时间，这个时间不包含加载组件的时间，即不包含处理 props、getInitialState，调用 componentWillMount 和 componentDidMount 等时间。
- Perf.printDOM(): 打印所有的 DOM 操作，例如“设置 innerHTML”和“移除节点”。

本节介绍了 React 本身带来的性能优化知识，并介绍了性能调试工具 React.addons.Perf。React 能够为开发者做到最大限度的性能保证，同时开发者也要遵循一些最佳实践来实现性能的稳定。下一节我们就从开发者的角度进行分析。

8.2 提升 React 应用性能的建议

通过前一节的介绍，我们了解到影响 React Redux 应用性能的是组件调用 `this.setState()` 方法时引发的“蝴蝶效应”。React 渲染真实的 DOM 节点的过程由两个主要过程组成。

- 对 React 内部维护的虚拟的 DOM 进行更新。
- 对前后两个虚拟的 DOM 进行对比，并将 diff 的结果应用于真实的 DOM 中。

前后两步极其关键，设想一下，如果虚拟的 DOM 更新很慢，那么重新渲染势必会很耗时。本节我们就针对此问题对症下药，来了解更多的性能优化技巧。

8.2.1 最大限度地减少重新渲染

为了提升 React 应用的性能，我们首先想到的就是最大限度地规避不必要的重新渲染。但是当状态发生变化时，重新渲染是 React 内部的默认行为，那么该如何保证不必要的渲染呢？

最先想到的一定是使用 `shouldComponentUpdate` 生命周期函数，它通过对比前后 `state/props` 是否发生变更，来决定组件是否需要重新渲染。

实际上，还有很多方式，开发者都可以给 React 发送“不需要渲染”的信号。

比如，无状态组件返回相同的 `element` 实例。如果 `render` 方法返回相同的 `element` 实例，React 会认为组件并没有发生变化。请参考以下代码。

```
class MyComponent extends Component {
  text = "";
  renderedElement = null;
  _render() {
    return <div>{this.props.text}</div>
  }
  render() {
    if (!this.renderedElement || this.props.text !== this.text) {
      this.text = this.props.text;
      this.renderedElement = _render();
    }
    return this.renderedElement;
  }
}
```

类似于上述逻辑，设想一下，当组件第一次实例化时，`render` 方法将会调用真正意义上的 `_render()` 完成渲染加载，并渲染 `this.props.text` 所提供的信息。同时更新 `this.text = this.props.text`，并记录渲染实例化的结果 `this.renderedElement`。这样一来，当组件再次以相同的 `this.props.text` 进行渲染时，便可以返回组件上一次实例化的结果 `this.renderedElement`。

熟悉 `lodash` 库的读者，可能会想到其带来的 `memoize` 函数，同样可以用来简化上述代码。

```
import memoize from 'lodash/memoize'

class MyComponent extends Component {
  _render = memoize((text) => <div>{text}</div>)
  render() {
    return _render(this.props.text)
  }
}
```

在之前介绍的高阶组件的基础上，我们不妨设想这样一类高阶组件：它能够细粒度地控制组件的渲染行为。比如，某个组件仅仅在某一项 `props` 变化时才会触发重新渲染。这样一来，开发者可以完全掌控组件渲染时机，更有针对性地进行渲染优化。

这样的方法有点类似于农业灌溉上的“滴灌”技术，它规避了代价昂贵的粗暴型灌溉，而是精准地定位需求，从而达到节约水资源的目的。

在社区中，优秀的 `recompose` 库恰好可以满足我们的需求。请参考如下代码。

```
@pure
class MyComponent1 extends Component {
  render() {
    ///...
  }
}
```

在引入 `recompose` 库的前提下，使用 `@pure` 修饰器，`MyComponent1` 组件只会在 `props` 改变的情况下进行渲染。你可能会想，这不就是无状态组件可以完成的吗？事实确实如此，再往下看，你就能明白更多的“黑魔法”。

```
@onlyUpdateForKeys(['prop1', 'prop2'])
class MyComponent2 extends Component {
  render() {
    ///...
  }
}
```

使用`@onlyUpdateForKeys`修饰器, `MyComponent2` 组件只有在 `prop1` 和 `prop2` 变化时才进行渲染; 否则, 其他的 `props` 发生任何改变, 都不会触发重新渲染。

如果你不喜欢使用修饰器, 则完全可以用熟悉的高阶组件来实现。

```
MyComponent = pure(MyComponent);
MyComponent = onlyUpdateForKeys(['prop1', 'prop2'])(MyComponent);
```

藏在 `onlyUpdateForKeys` 背后的“黑魔法”其实并不难理解, 只需要在高阶组件中调用 `shouldComponentUpdate` 方法, 在 `shouldComponentUpdate` 方法中比较对象由完整的 `props` 转为传入的指定 `props` 即可。有兴趣的读者, 可以翻阅 `recompose` 源码进行了解, 其实思路即是如此。

8.2.2 Redux connect 方法隐藏的性能优化思想

当我们在使用 `Redux` 的时候, 一定对 `connect` 方法并不陌生。它实际上也是一个高阶组件: 从 `store` 中提取信息, 利用 `React context` 特性进行传递, 并在 `state` 发生变化时调用其参数 `mapStateToProps` 来进行响应, 最终触发相关组件的渲染。

在这个过程中, 只有当相关数据发生变化时, 相连通的组件才会被重新渲染。比如:

```
connect(state => ({
  prop1: state.prop1
})) (componentA)
```

`componentA` 组件只有在 `prop1` 发生变化时, 才会触发重新渲染。但是对这种所谓的“发生变化”的判断, 采用的是浅比较, 当浅比较结果不同时, 则认定发生了变化。如果 `this.prop1` 是简单类型, 这是没有任何问题的。

```
connect(state => ({
  hasSomething: this.prop1 === 5 // true===true
})) (SomeComponent)
```

但是对于复杂类型, 比如在 `mapStateToProps` 中导出一个对象, 则比较的是内存地址, 而不是“值是否相等”。

```
connect(state => ({
  computedData: {
    height: state.height,
    width: state.width
  }
})) (SomeComponent)
```


在上面 `connect` 的 `mapStateToProps` 方法中，我们根据 `state` 计算并衍生出 `computedData` 对象，将其传递给相关组件。因为 `computedData` 是一个对象类型，每次执行 `mapStateToProps` 后都会生成一个新对象，即使对象值相等，也会进行 `SomeComponent` 组件渲染。在这种情况下，仍然会发生一些不必要的重复渲染。幸运的是，我们还是有比较成熟的方法来规避这样的问题的。比如先前提到的 `reselect` 库，它的任务就是完成由 `state` 到衍生数据的计算。另外，这种计算是有记忆的。

```
import {createSelector} from 'reselect'

const selectComputedData = createSelector(
  state => state.height,
  state => state.width,
  (height, width) => ({
    height,
    width
  })
)

connect(state => ({
  computedData: selectComputedData(state)
}))(SomeComponent)
```

事实上，在不使用第三方库 `reselect` 的情况下，依靠 `Redux` 的已有特性，也能够解决问题。`Redux` 给出的 `connect` 方法有很多“不为人知”的秘密武器。官方对 `connect` 的语法应用解读为：

```
connect(
  [mapStateToProps],
  [mapDispatchToProps],
  [mergeProps],
  [options]
)
```

秘密就在最后一个参数 `options` 中：

```
[options] = {
  pure = true,
  areStatesEqual = strictEqual,
  areOwnPropsEqual = shallowEqual,
  areStatePropsEqual = shallowEqual,
  areMergedPropsEqual = shallowEqual,
  ...extraOptions
}
```


在 `options` 中, 支持自定义 `areStatesEqual` 函数, 以便进行前后两次 `state` 比较。`areStatesEqual` 函数接收两个参数, 第一个是前一个 `state`, 第二个是后一个 `state`。如果该函数的返回值为 `true`, 那么 `mapStateToProps` 方法便不再执行, 因为规避了不必要的渲染。

结合场景, 我们可以通过以下代码来实现。

```
connect(
  state => ({
    computedData: {
      height: state.height,
      width: state.width
    }
  }),
  [mapDispatchToProps],
  [mergeProps],
  {
    areStatesEqual: (prev, next) => {
      return (
        prev.height === next.height &&
        prev.width === next.width
      );
    }
  }
)(SomeComponent)
```

很多开发者在使用 `Redux` 开发应用时, 在大多数情况下对 `API` 和特性的使用都很集中。一般来说, `connect` 方法用到 `mapStateToProps` 和 `mapDispatchToProps` 两个参数基本上就能够解决问题。但是在开发精力充沛的情况下, 对更多的特性进行探索很有必要。通过探索, 思考这些优秀的状态管理方案及类库的设计, 对综合能力的提升非常关键。同时, 这些特性很多都是为性能而考虑的。具体来看:

除 `mapStateToProps` 和 `mapDispatchToProps` 常用以外, 第三个参数 `mergeProps` 也很有意义。作为高阶组件, 所有传入 `connect` 函数的参数只有一个目的, 那就是生成一个对象, 这个对象将作为目标组件的 `props` 出现。顾名思义, `mergeProps` 就是将注入目标组件的 `props` 的对象进行合并。合并的来源有三个, 即 `mergeProps` 接收的三个参数——`stateProps`、`dispatchProps` 和 `ownProps`。

`stateProps` 即 `mapStateToProps` 函数的返回值, 它是由 `state` 映射而来的; `dispatchProps` 即

`mapDispatchToProps` 函数的返回值，简单地理解为它是一系列 `action creator` 组成的对象；`ownProps` 则是从目标组件本身继承而来的，即被父组件传递进来的 `props` 部分。

默认地，`mergeProps` 会将这三个来源进行合并，得到的对象将作为最终注入目标组件的 `props`。

```
{ ...ownProps, ...stateProps, ...dispatchProps }
```

如果开发者对 `mergeProps` 进行了重写，则可以自定义组合最终的 `props` 结果。

`connect` 的第四个参数 `options` 是对象类型，其中包括 `pure`（是布尔值）和都返回一个布尔值的四个函数。它们的共同作用就是决定是否触发组件重新渲染。

`pure` 默认为 `true`，如果开发者将其设置为 `false`，那么 `connect` 高阶组件将会跳过所有优化步骤，且 `options` 的四个函数将被忽略。因此，很少有场景将 `pure` 设置为 `false`。

上面提到的 `mergeProps` 函数生成的最终 `props` 结果会与前一次生成的 `props` 结果进行对比，这就是前面提到的“当 `state` 发生变化时，`connect` 采用浅比较来判断是否需要触发重新渲染”。在源码中，具体比较函数的关键部分如下（有删减）：

```
shallowEqual(objA, objB) {  
  const keysA = Object.keys(objA)  
  const keysB = Object.keys(objB)  
  if (keysA.length !== keysB.length) return false  
  
  for (let i = 0; i < keysA.length; i++) {  
    if (!hasOwn.call(objB, keysA[i]) ||  
        !is(objA[keysA[i]], objB[keysA[i]])) {  
      return false  
    }  
  }  
}
```

这是很典型的浅比较，即在两个目标对象属性长度相等的情况下，对目标对象 `objA` 进行遍历，检查在 `objB` 中是否存在同样的值。

8.2.3 inline function 的反模式

我们需要注意一种非常重要的“反模式”。当使用 `render` 方法时，要留意在 `render` 方法内创

建的函数或者数组等，这些创建操作可能是显式的，也可能是隐式的。因为这些新生成的函数或数组，当量大时会造成一定的性能负担。同时，render 方法经常被反复执行多次，也就是说，总有新的函数或数组被创建，这样将造成内存无意义的开销。通常对性能更友好的做法是它们只需被创建一次，而不是每次渲染时都被创建。比如：

```
render() {
  return <MyInput onChange={this.props.update.bind(this)} />;
}
```

或者

```
render() {
  return <MyInput onChange={() => this.props.update()} />;
}
```

使用 bind 方法，每次渲染时都会创建一个新函数，对内存造成不必要的消耗。在完成 this 绑定的情况下，提倡的做法是：

```
onChange() {
  this.props.doUpdate()
}

render() {
  return <MyInput onChange={this.onChange} />;
}
```

我们同样可以在 redux.connect 的 mapDispatchToProps 方法中完成函数的绑定，进而避免在 render 内进行多次创建，以此来规避问题。

```
connect(null, (dispatch, ownProps) => {
  onChange: event => {
    dispatch(actions.updateValue(event.target.value))
  }
})(SomeComponent)
```

```
class SomeComponent extends Component {
  render() {
    return <MyInput onChange={this.props.onChange} />;
  }
}
```

对于在 render 方法内创建数组或其他类型的情况，也存在类似的问题：

```
render() {  
  return <SubComponent items={this.props.items || []}/>  
}
```

这样做会在每次渲染且 `this.props.items` 不存在时都创建一个空数组。更好的做法是：

```
const EMPTY_ARRAY = [];  
render() {  
  return <SubComponent items={this.props.items || EMPTY_ARRAY}/>  
}
```

8.3 使用 PureComponent 保证开发性能

React 15.3 具有 `React.PureComponent` 特性，取代了之前的 `pure-render-mixin`。在本文中，我们将讨论 `PureComponent` 究竟是什么，为什么它能带来性能上的保障，以及在什么场景下应该使用 `PureComponent` 等问题。

其实 `PureComponent` 大体与 `Component` 相同，唯一不同的地方是 `PureComponent` 会自动帮助开发者使用 `shouldComponentUpdate` 生命周期方法。也就是说，当组件 `state` 或者 `props` 发生变化时，正常的 `Component` 都会自动进行重新渲染，在这种情况下，`shouldComponentUpdate` 默认都会返回 `true`。但是 `PureComponent` 会先进行对比，即比较前后两次 `state` 和 `props` 是否相等。需要注意的是，这种对比是浅比较。

使用 `PureComponent` 声明组件非常简单：

```
import React, { PureComponent } from 'react'  
  
class Example extends PureComponent {  
  render() {  
    // ...  
  }  
}
```

我们已经了解到，使用 `PureComponent` 声明的组件，会自动帮助开发者在 `shouldComponentUpdate` 方法中进行对比。实现代码如下：

```
function shallowEqual (objA: mixed, objB: mixed) {  
  if (is(objA, objB)) {  
    return true;  
  }
```

```
}

if (typeof objA !== 'object' || objA === null ||
    typeof objB !== 'object' || objB === null) {
  return false;
}

const keysA = Object.keys(objA);
const keysB = Object.keys(objB);

if (keysA.length !== keysB.length) {
  return false;
}

for (let i = 0; i < keysA.length; i++) {
  if (
    !hasOwnProperty.call(objB, keysA[i]) ||
    !is(objA[keysA[i]], objB[keysA[i]])
  ) {
    return false;
  }
}

return true;
}
```

基于以上代码，我们总结出使用 `PureComponent` 需要注意如下细节。

- 既然是浅比较，也就是说，当与前一个 `props` 和 `state` 比对时，如果比较对象是 JavaScript 基本类型，那么会对其值是否相等进行判断；如果比较对象是复杂的 JavaScript 类型，比如 `object` 或者 `array`，则会判断其引用是否相同，而不会进行值比较。
- 开发者需要避免共享（`mutate`）带来的问题。

注意：不要在 `object` 或者 `props` 中进行共享这一“万恶操作”。如果在一个父组件中对 `object` 进行了共享操作，若子组件依赖此数据，且采用 `PureComponent` 声明，那么子组件将无法进行更新。尽管 `props` 中的某一项值发生了变化，但是它的引用并没有发生变化，所以 `PureComponent` 的 `shouldComponentUpdate` 也就返回了 `false`。更好的做法是在更新 `props` 或 `state` 时，返回一个新的对象或数组。

下面举一个例子来理解。假如父组件有一个 `render` 方法和一个 `click` 处理方法。当点击按钮之后，试图新增加一个条目到 `items` 中。

```
handleClick() {  
  let {items} = this.state;  
  
  items.push('new-item');  
  this.setState({ items });  
}  
  
render() {  
  return (  
    <div>  
      <button onClick={this.handleClick} />  
      <ItemList items={this.state.items} />  
    </div>  
  )  
}
```

如果 `ItemList` 是使用 `PureComponent` 声明的组件：

```
class ItemList extends PureComponent {  
  render() {  
    // ...  
  }  
}
```

当点击事件发生后，尽管 `this.state.items` 值发生了变化，但是它的引用地址并没有改变，因此点击按钮后，不会引起 `ItemList` 的渲染。

基本类型的比较或 JavaScript 对象的引用地址的比较，都是相对高效且容易实现的操作，计算成本非常低廉。这也是 `PureComponent` 被广泛使用的基础。设想一下，如果应用组件非常复杂，或者包含一个具有很长 `list` 的组件，若只是其中一个子组件发生了变化，那么使用 `PureComponent` 进行对比，有选择性地渲染，一定比所有列表数据都被重新渲染划算很多。

为此，我们可以简单做一个采用 `PureComponent` 和不采用 `PureComponent` 进行性能对比的测试。假如在页面中需要渲染非常多的用户信息，所有的用户信息都被维护在一个 `users` 数组当中，数组的每一项都是一个 JavaScript 对象，表示一个用户的基本信息。`User` 组件负责渲染每一个用户的信息内容。


```
import User from './User'

const Users = ({users}) =>

  <div>

    {users.map(user => <User {...user} />)}

  </div>
```

在使用 Redux 的情况下，有对应的 reducer 函数，代码如下：

```
const users = (state, action) => {
  if (action.type === 'CHANGE_USER_1') {
    return [action.payload, ...state.slice(1)]
  }
  return state
}
```

当触发 'CHANGE_USER_1' action 时，表示需要对 users 的第一项，即第一个用户的信息进行更新。为了保证 reducer 函数的纯净，我们在 reducer 逻辑中返回了一个全新的数组。

这样做存在的问题是：users 数组作为 User 组件的 props 出现，当数组的第 K 项发生变化时，users 数组即发生变化，所有的 User 组件都会进行 reconciliation，即使非 K 项并没有发生变化，也会引起不必要的渲染。

在测试中，我们渲染了一个有 200 项数据的数组，代码如下：

```
const arraySize = 200;
const getUsers = () =>
  Array(arraySize)
    .fill(1)
    .map((_, index) => ({
      name: 'John Doe',
      hobby: 'Painting',
      age: index === 0 ? Math.random() * 100 : 50
    }));
```

注意：在 getUsers 方法中，对 age 属性进行了判断，保证每次调用时 getUsers 返回的数组只有第一项的 age 属性不同，其余的全部为 50。在测试组件中，在 componentDidUpdate 中保证数组将会触发 400 次重新渲染，并且每一次只改变数组第一项的 age 属性，其他的均保持不变。

```
const repeats = 400;
componentDidUpdate() {
```

```

    ++this.renderCount;
    this.dt += performance.now() - this.startTime;
    if (this.renderCount % repeats === 0) {
      if (this.componentUnderTestIndex > -1) {
        this.dts[componentsToTest[this.componentUnderTestIndex]] = this.dt;
        console.log(
          'dt',
          componentsToTest[this.componentUnderTestIndex],
          this.dt
        );
      }
      ++this.componentUnderTestIndex;
      this.dt = 0;
      this.componentUnderTest = componentsToTest[this.componentUnderTestIndex];
    }
    if (this.componentUnderTest) {
      setTimeout(() => {
        this.startTime = performance.now();
        this.setState({ users: getUsers() });
      }, 0);
    }
    else {
      alert(`
        Render Performance ArraySize: ${arraySize} Repeats: ${repeats}
        Functional: ${Math.round(this.dts.Functional)} ms
        PureComponent: ${Math.round(this.dts.PureComponent)} ms
        Component: ${Math.round(this.dts.Component)} ms
      `);
    }
  }
}

```

下面对三种组件声明方式进行对比。

(1) 函数式方式

```

export const Functional = ({ name, age, hobby }) => (
  <div>
    <span>{name}</span>
    <span>{age}</span>
    <span>{hobby}</span>
  </div>
)

```

```
    </div>
  );
}
```

（2）PureComponent 方式

```
export class PureComponent extends React.PureComponent {
  render() {
    const { name, age, hobby } = this.props;
    return (
      <div>
        <span>{name}</span>
        <span>{age}</span>
        <span>{hobby}</span>
      </div>
    );
  }
}
```

（3）经典 class 方式

```
export class Component extends React.Component {
  render() {
    const { name, age, hobby } = this.props;
    return (
      <div>
        <span>{name}</span>
        <span>{age}</span>
        <span>{hobby}</span>
      </div>
    );
  }
}
```

在使用 `PureComponent` 声明的组件中，会自动在触发渲染前后进行 `{name, age, hobby}` 对象值比较。如果没有发生变化，则 `shouldComponentUpdate` 返回 `false`，以规避不必要的渲染。因此，使用 `PureComponent` 声明的组件性能明显优于其他方式。在不同的浏览器环境下，可以得出：

- 在 Firefox 下，PureComponent 收益 30%。
- 在 Safari 下，PureComponent 收益 6%。
- 在 Chrome 下，PureComponent 收益 15%。

实际上，我们通过定义 `changedItems` 来表示变化数组的元素，`array` 表示所需渲染的数组。`changedItems.length/array.length` 的比值越小，表示数组中变化的元素越少，`React.PureComponent` 涉及的性能优化也越有必要实施，因为 `React.PureComponent` 通过浅比较规避了不必要的更新过程，而浅比较自身的计算成本一般都不值一提，节约了成本。

当然，`PureComponent` 也不是万能的，尤其是它的浅比较，需要开发者格外注意。所以在特定情况下，开发者根据需求自己实现 `shouldComponentUpdate` 中的比较逻辑，将是更高效的选择。

8.4 Redux 中间件和 Web Worker

在本书第 4 章中我们了解了 Redux 的设计理念，探索了中间件奥秘。在本章前面也讨论了关于 React 性能优化的事项。其实中间件和性能优化也可以相互结合，形成一个完美的性能优化复用思路。当然，还需要一个重要的角色——Web Worker。

随着 Web 应用复杂度的提升，数据计算压力激增，当 `reducer` 函数变得臃肿不堪时，我们可以拆分 `reducer` 使代码看上去更加舒服。但是对于一些庞大的计算需求，其带来的性能瓶颈永远无法消除。

我们知道 JavaScript 是单线程异步语言，可能也了解过黄金 60fps，即每一帧绘制留给 JavaScript 代码执行的时间为 16ms（甚至更少）。那么一旦 `reducer` 计算时间过长，就必然会会影响浏览器渲染，性能问题直接显现在用户面前。

这就引出了 Web Worker 的概念。2008 年 W3C 制定了第一个 HTML 5 草案，HTML 5 承载了越来越多崭新的特性和功能，其中最重要的便是对多线程的支持。在 HTML 5 中提出了 Web Worker 的概念，并且规范了 Web Worker 的三大主要特征：

- 能够长时间运行（响应）。
- 理想的启动性能。

- 理想的内存消耗。

Worker 线程可以执行任务而不干扰用户界面，那么是否可以将 Redux 下 reducer 复杂的计算部分放进 Worker 线程中处理呢？答案是肯定的。

我们先来看一下经典的 Redux 流程图，如图 8-4 所示。

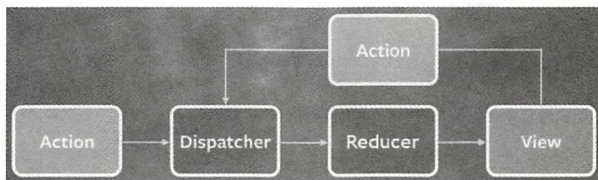


图8-4 经典的Redux流程图

通过本书第 4 章介绍的中间件知识，我们可以设计加入 Worker 计算，如图 8-5 所示。

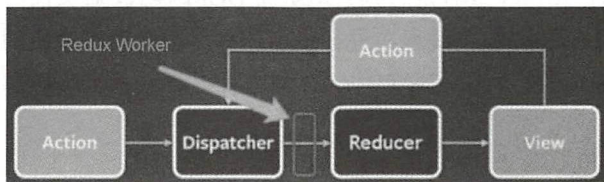


图8-5 Worker中间件示意图

当然，有了思路，还需要在实战中演练。

下面演示使用“ N 皇后算法”模拟大型计算，并任意设置 n 值，增加计算的耗时。不理解此算法也没有关系，只需清楚 N 皇后算法的计算耗时时长和 n 值相关即可—— n 值越大，计算成本越高。

除了 N 皇后算法，在页面中还运行了如下几个模块，以实现复杂的渲染逻辑操作（见图 8-6）。

- 一个实时显示计数的 Counter 模块，计数数值每秒增加 1。
- 一个每 500 毫秒更新一次背景颜色的 Blinker 模块。
- 一个永久往复左右运动的 Slider 模块。
- 一个每 16 毫秒翻转 5 度的 Spinner 模块。

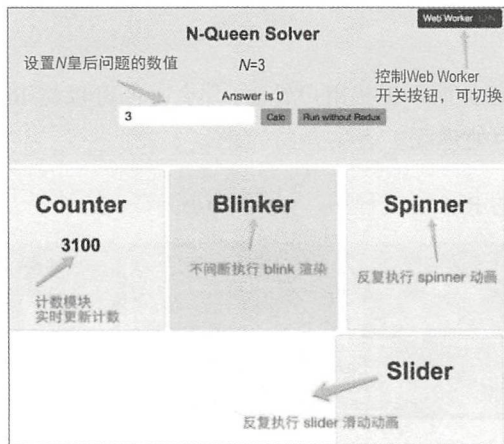


图8-6 页面模块示意图

这些模块或者由动画组成, 或者定时频繁地更新 DOM 样式, 对页面流畅渲染的要求很高。在正常情况下, 由于 JavaScript 主线程在进行皇后计算, 因此这些模块的脚本逻辑渲染需求都将被影响, 而造成页面卡顿。在卡顿过程中, 任何事件 (如点击、按键等) 都无法被浏览器响应。

如果把 N 皇后计算迁移至 Worker 线程中, 则丝毫不会影响页面的渲染和绘制, 先前所有的性能问题都可以得到很好的解决。

我们试图将计算过程交给 Worker 线程, 为了更加通用, 可以把 Web Worker 的应用抽象出一个公共库: `redux-worker`, 并包装为 Redux 的中间件, 所有的 React Redux 应用都可以采用中间件的思想, 无侵入使用。

```
import { applyWorker } from 'redux-worker';

const enhancerWithWorker = compose(
  applyMiddleware(thunk, logger),
  applyWorker(worker)
);

const store = createStore(rootReducer, {}, enhancerWithWorker);
```

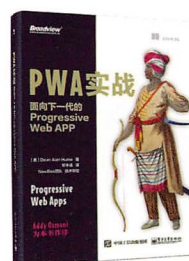
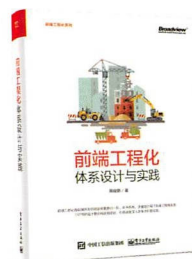
`redux-worker` 提供的 `applyWorker` 接口配合 Redux `compose` 方法的原理, 通过第 4 章内容我们已经有所了解。`applyWorker` 逻辑并不难理解, 全部源码以及演示样例都可以在本书配套的代码仓库中找到。相信随着前端的发展, Web Worker 的使用场景会越来越多, 结合 React Redux 的思路也会越来越重要。

8.5 本章小结

性能优化是前端开发中一个永恒的话题，不同框架之间的性能对比也一直是开发者关注的方面。性能涉及方方面面，如前端工程化、浏览器解析和渲染、比较算法等。本章主要介绍了 React 框架在性能上的优劣、虚拟的 DOM 思想，以及在开发 React 应用时需要注意的性能优化环节和手段。

也许不是每个应用都会面临性能的问题，如同社区中所说的：“过早地进行性能优化是毫无必要的，但是开发者在性能优化方面的积累却要时刻先行。”同时，优化手段也在与时俱进，不断更新，需要开发者时刻保持学习。

好书分享



世界的数字化进程正在加速，我们也进入了体验至上的新移动时代。在连接人和信息世界的所有技术中，前端开发作为直接影响用户体验的关键一环，正在迅速发展和变革，前端工程师也成为产品研发团队里关键拼图之一，决定了产品迭代的成败。本书以 React 为中心，在讲解相关技术栈的同时，深刻剖析了隐藏在其后的编程思想，希望更多的开发者能够以这本书为起点，深入把握前端开发技术，活学活用，打造极致的用户体验，为新移动时代创造更多优秀的产品。

—— 百度公司副总裁，沈抖

本书针对 React 进行了专题研究，其中还包含 Redux 用法的详细介绍、源码解读、中间件的实现原理，以及前后端同构的解决方案（即服务器渲染）等内容，非常适合初学者进阶学习 React 的相关知识，掌握实战技能。建议各位读者按照源码上机练习，以达到更好的学习效果。

—— 资深 JavaScript 语言专家、知名技术博客作者，阮一峰

很高兴能够看到这本书，它系统讲解了 Redux 和同构技术，是一本在垂直领域中非常优秀的专业书籍，从入门到组件通信，再到源码，都给出了非常详细的解读。本书简明扼要，重视实践，尤其适合初学者。学会 Redux 可以让你在前端开发中更加游刃有余，同构开发对于拓宽前端开发领域也有着极其重要的意义，建议大家深入学习。

—— Node.js 布道者、Cnode 社区管理员，狼叔（i5ting）

React 以及 Redux 是目前非常流行的技术栈，本书深入 React 技术，涉及 React 的方方面面内容，从入门到高阶实例，从状态管理到同构应用技巧，无论是在技术实现原理上，还是在实战经验上，都能帮助读者对 React 形成全貌理解。无论你是 React 新手还是资深工程师，相信都能从本书中获得启发。

—— 新浪移动前端技术专家，付强（小燚）

本书由浅入深，从 React 涉及的基本概念开始不断延伸，不仅分场景覆盖了以 React、Redux、Next.js 等为核心的组件化开发流程及工程优化策略，还深入源码解读了技术细节，并通过对设计思路的阐述来帮助读者增强系统性的理解和认知。相信本书可以帮助广大开发者更好地掌握 React 体系的精髓，创造出体验更佳的产品。

—— 百度资深前端工程师、知乎知名博主，顾轶灵



博文视点Broadview



新浪微博
weibo.com

@博文视点Broadview



策划编辑：孙奇俏
责任编辑：葛娜
封面设计：吴海燕

欢迎投稿

邮箱：sunqq@phei.com.cn

微信：sunqiqiao

上架建议：前端开发/React

ISBN 978-7-121-34554-8



9 787121 345548 >

定价：79.00元